M6713 User's Manual

The M6713 User's Manual was prepared by the technical staff of Innovative Integration on March 20, 2006.

For further assistance contact:

Innovative Integration
2390-A Ward Avenue,
Simi Valley, California 93065

PH:(805) 578-4260
FAX:(805) 578-4225
email:techsprt@innovative-dsp.com
Website:www.innovative-dsp.com

This document is copyright 2005 by Innovative Integration. All rights are reserved.

VSS\Distributions\M6713\Documentation\Manual\M6713.book

*CHAPTER 1 Introduction **11***

Introduction **11**
Finding detailed information on Pismo and the Host Libraries **12**

*CHAPTER 2 Installation **15***

Host Hardware Requirements **15**
Software Installation **15**
Tools Registration **18**
Hardware Installation **19**
Code Composer Studio v2.x Setup **21**
Code Composer Studio v3.x Setup **24**
Borland Builder Setup and Use **27**

*CHAPTER 3 About the Baseboard **31***

DSP Baseboard Hardware Features **31**
Digital Signal Processor **32**
The Pismo Class Library **33**
Analog I/O Streams **35**
Interrupt Handling **42**
EDMA and QDMA Handling **45**

*CHAPTER 4 Building a Target DSP Project **51***

Building a Target DSP Project **51**
Writing a Program **57**
Host Tools for Target Application Development **57**
Edit-Compile-Test Cycle using Code Composer Studio **58**
Anatomy of a Target Program **60**
Example Programs **61**
The Next Step: Developing Custom Code **62**

*CHAPTER 5 Servo Applications **65***

Using the Servo Class **66**
A Servo Tutorial **69**

*CHAPTER 6 Communication to the Host **73***

Overview **73**
CPU Busmastering Interface **74**
C++ Terminal I/O **76**

*CHAPTER 7 Developing Host Code **79***

The BoardLib library **79**
The M6713 in the Host Environment **80**
Host Example Program for the M6713 Baseboard **83**

*CHAPTER 8 Applets **89***

Registration Utility (NewUser.exe) **89**
ReserveMemoryDsp **90**
Target Download Utility (M6713Download.exe) **90**
Logic Download Utility (M6713LogicLoader.exe) **90**
Logic Update Utility (M6713VsProm.exe) **91**
JTAG Diagnostic Utility (JtagDiag.exe) **91**
Demangle Utility (Demangle.exe) **91**
COFF Section Dump Utility (CoffDump.exe) **91** **92**
Target Project Copy Utility (CopyCcsProject.exe) **92**
Binary File Viewer Utility (BinView.exe) **92**
DEF Conversion Utility (DefConvert.exe) **93**
Scan Path Diagnostic Utility (JtagScanpath.exe) **93**
RtdxTerminal “The Terminal Emulator” **93**

*CHAPTER 9 M6713 Hardware **101***

M6713 Hardware Functions **101**
Memory Map **102**
M6713 Hardware Initialization Requirements **105**
External Memory **106**
M6713 OMNIBUS **107**
FPDP Port I/O Expansion **109**
‘C6713 McBSP Serial Ports **113**
PCI Interface **114**
Timers **116**
Digital I/O **119**
Interrupts **122**
External Inputs for Clocks and Interrupts **126**
Multi-Card Timing Synchronization **127**
JTAG Test Bus **129**
Power Requirements **129**
Updating the M6713 logic **129**
Making Custom Logic **130**

*CHAPTER 10 Troubleshooting **133***

Initialization Problems **133**

Borland C++Builder Problems **133**
DSP Hardware Problems **135**

*CHAPTER 11 Appendices **137***

Connector pinouts **137**
Board Layout Drawing (Rev B) **152**

TABLE 1. Typographic Conventions 13
TABLE 2. ‘C6713 DSP EMIF Control Register Initialization Values 32
TABLE 3. Device Driver and Stream Classes 35
TABLE 4. Stream object Clock Methods 39
TABLE 5. Stream object Pretrigger Methods 40
TABLE 6. Stream object Start Trigger Methods 40
TABLE 7. Stream object Stop Trigger Methods 40
TABLE 8. Stream object Retrigger Methods 41
TABLE 9. Interrupt Lock Classes 44
TABLE 10. Pismo Example Programs 61
TABLE 11. The Servo Class Virtual Function 66
TABLE 12. Servo Time Measurements 68
TABLE 13. M6713 DSP External Memory Map 105
TABLE 14. M6713 Bus Control Register Initialization Values 106
TABLE 15. M6713 I/O Bus Memory Mapping 107
TABLE 16. I/O Bus Power Ratings 109
TABLE 17. FPDP Memory Map 113
TABLE 18. FPDP Rx Configuration Register. 113
TABLE 19. FPDP Tx Configuration Register. 113
TABLE 20. PCI DMA Rates Summary 115
TABLE 21. DDS Post-dscaling Control Register (Module 0 0x8020002C, Module 1 0x80200030) 118
TABLE 22. AD9851 Control Registers 118
TABLE 23. Timer0 and DDS Pin Output Sources (Module 0 0x80200034, Module 1 0x80200038) 119
TABLE 24. Digital I/O Control Registers 120
TABLE 25. Digital IO Port Control Register Bit Definitions 121
TABLE 26. Digital I/O Port Timing Parameters 122
TABLE 27. External Interrupt Input Control Register Addresses 122
TABLE 28. External Interrupt Control Register Bit Definitions 123
TABLE 29. External Interrupt Status and Acknowledge Register Addresses 125
TABLE 30. Interrupt Burst Counting Control Register 126
TABLE 31. DMA Address for Interrupt Control 126
TABLE 32. Interrupt Access Counting Control Registers 126
TABLE 33. External Clock Input Termination Jumper Settings 127
TABLE 34. SyncLink 0 Signal Selection Register Bit Definitions 128
TABLE 35. SyncLink 1 Signal Selection Register Bit definitions 128
TABLE 36. SyncLink 2 Signal Selection Register Bit definitions 128
TABLE 37. ClockLink Signal Selection Register Bit Definitions 129
TABLE 38. OMNIBUS Bus Connectors 140
TABLE 39. I/O Module Bus Connectors 141
TABLE 40. Digital I/O Connector 142
TABLE 41. FPDP JH1 Tx Port Connector 144

TABLE 42. FPDP JH2 Rx Port Connector 146
TABLE 43. SyncLink Connector 147
TABLE 44. DSP Serial Port Connector 148
TABLE 45. DSP JTAG Debugger Connector 149
TABLE 46. Power Test Connector 150
TABLE 47. JP17 Interface Logic (Spartan3) JTAG Connector 151

FIGURE 1. Some Classes in Malibu 80
FIGURE 2. Terminal Emulator Applet 94
FIGURE 3. RtdxTerminal File Menu 95
FIGURE 4. RtdxTerminal DSP Menu 96
FIGURE 5. RtdxTerminal Form Menu 96
FIGURE 6. RtdxTerminal Help Menu 97
FIGURE 7. RtdxTerminal Options 98
FIGURE 8. M6713 Block Diagram 102
FIGURE 9. FPDP Overview. 110
FIGURE 10. FPDP Timing Diagrams for ALL Data Framing Types. 111
FIGURE 11. Standard FPDP Timing Diagram For Single Frame And Repeated Frame Data. 112
FIGURE 12. PCI Interface Block Diagram 114
FIGURE 13. PCI Packet Format 115
FIGURE 14. DDS and Post-scaler 117
FIGURE 15. Digital IO Port Block Diagram 120
FIGURE 16. Digital I/O Port Timing 122
FIGURE 17. JP12 SyncLink Connector Pin Orientation 147
FIGURE 18. JP14, JP15 DSP Serial Port Connector 148
FIGURE 19. JP16 DSP JTAG Debugger Connector 149
FIGURE 20. JP18 Power Test Connector 150
FIGURE 21. JP17 Interface Logic (Spartan3) JTAG Connector 151

CHAPTER 1

Introduction

Introduction

When making reference to the “baseboard” in this manual, we will be referring to the M6713 single-board computer baseboard.

What is the M6713? The M6713 is Innovative Integration’s PCI plug-in baseboard architecture that integrates modularized, high-performance analog and digital peripherals with a high-performance DSP and peripheral cores. The M6713 is equipped with a 64-bit, 66 MHz PCI bus interface allowing high speed data transfers with the host.

The baseboard includes an onboard TMS320C6713 DSP with 128 MB cached SDRAM. The DSP accesses to the entire baseboard peripheral complement directly as memory-mapped devices. The baseboard supports DMA over the PCI bus at speeds up to 512 MB/sec, a packet-message-based bus interface, two variable-function Omnibus I/O module sites, dedicated Front Panel Data Ports (FPDP) and SyncLink buses for inter-board connectivity, a precision DDS timebase to serve as an accurate, programmable clock source from DC-25 MHz and a programmable digital I/O port. Because of the onboard DSP, the baseboard is capable of performing data collection, servo or other real-time processing and data movement automatically without Host PC CPU involvement.

What is C++ Builder? C++ Builder is a general-purpose code-authoring environment suitable for development of Windows applications of any type. The M6713 host software library is a C++ class library that provides classes for control of the board hardware, communication with the target board over the PCI bus link, and support classes. By using standard C++ classes, multiple host platforms can be supported with a single code base, sharing identical functionality and interfaces.

What is Microsoft MSVC? MSVC is a general-purpose code-authoring environment suitable for development of Windows applications of any type.

What kinds of problems can I solve? Embedded data acquisition, servo control, stimulus-response and signal processing jobs are easily solved with the M6713 baseboard using the supplied Pismo software. There are a wide selection of peripheral devices available as plug-in Omnibus modules, for many types of signals from DC to RF frequency applications or audio processing. Additionally, multiple baseboards cards can be used for a large channel or mixed requirement systems and data acquisition cards from Innovative can be integrated with Innovative's other DSP or data acquisition baseboards (such as ChicoPlus) for high-performance signal processing.

Why do I need to use the M6713 baseboard? One of the biggest issues in using the personal computer for data collection, control, and communications applications is the relatively poor real-time performance associated with the system. Despite the high computational power of the PC, it cannot reliably respond to real-time events at rates much faster than a few hundred hertz. The PC is really best at processing data, not collecting it. In fact, most modern operating systems like Windows are simply not focused on real-time performance, but rather on ease of use and convenience. Word processing and spreadsheets are simply not high-performance real-time tasks.

The solution to this problem is to provide specialized hardware assistance responsible solely for real-time tasks. Much the same as a dedicated video subsystem is required for adequate display performance, dedicated hardware for real-time data collection and signal processing is needed. This is precisely the focus of the M6713 baseboard – a high performance, state-of-the-art, dedicated digital signal processor coupled with real-time data I/O capable of flowing data via the PCI bus.

Finding detailed information on Pismo and the Host Libraries

Information on Pismo and the host BoardLib is available in a variety of forms:

- On-line Help
- Innovative Integration Technical Support
- Innovative Integration Web Site (**www.innovative-dsp.com**)

Online Help

The on-line help system for the Host support classes to control the M6713 are contained in the `Malibu.hlp` help files placed into the `Program Files\Innovative\Manuals` directory tree during the default installationIt provides detailed information about the C++ control objects and usage examples. An equivalent version of this help file in HTML help format is also provided: `Malibu.chm`, for use within the MSVC context. The documentation for the target-side (DSP) Pismo C++ libraries is available in `M6713Pismo.hlp`/`M6713Pismo.chm`.

Innovative Integration Technical Support

Innovative includes a variety of technical support facilities as part of the M6713 toolset. Telephone hotline supported is available via

Hotline (805) 520-3300 8:00AM-5:00 PM PST.

Alternately, you may e-mail your technical questions at any time to:

techsprt@innovative-dsp.com.

Innovative Integration Web Site

Additional information on the Innovative product family and the M6713 DSP board is available via Innovative Online at www.innovative-dsp.com

Typographic Conventions. This manual uses the typefaces described below to indicate special text.

TABLE 1. Typographic Conventions

Typeface	Meaning
Monospace Type	Monospace type represents text as it appears onscreen or in code. It also represents anything you must type.
Boldface	Boldface words in text or code listings represent reserved words or compiler options.
Italics	Italicized words in text represent C++ Builder identifiers, such as variables or type names. Italics are also used to emphasize certain words, such as new terms.
`Keycaps`	This typeface indicates a key on your keyboard. For example, “Press `Esc` to exit a menu”.

CHAPTER 2 Installation

Thank you for purchasing from Innovative Integration! We appreciate your business. This chapter describes the software and hardware installation procedure for Windows 2K or XP. Development under other versions of Windows are not supported at this time, though drivers are provided to allow runtime operation under Win9x, and WinME.

Do *NOT* install the hardware card into your system at this time. This will follow the software installation.

Host Hardware Requirements

The software development tools require an IBM or 100% compatible Pentium IV-class or higher machine for proper operation. An Intel-brand processor CPU is *strongly recommended*, since AMD and other "clone" processors are not compatible with the Intel MMX and SIMD instruction-set extensions which the Malibu Host library utilizes extensively to improve processing performance within a number of its components. The host system must have at least 128 Mbytes of memory (256MB recommended), 100 Mbytes available hard disk space, and a CDROM drive. Windows 2000 or XP (referred to herein simply as *Windows*) is required to run the developer's package software, and are the target operating systems for which host software development is supported.

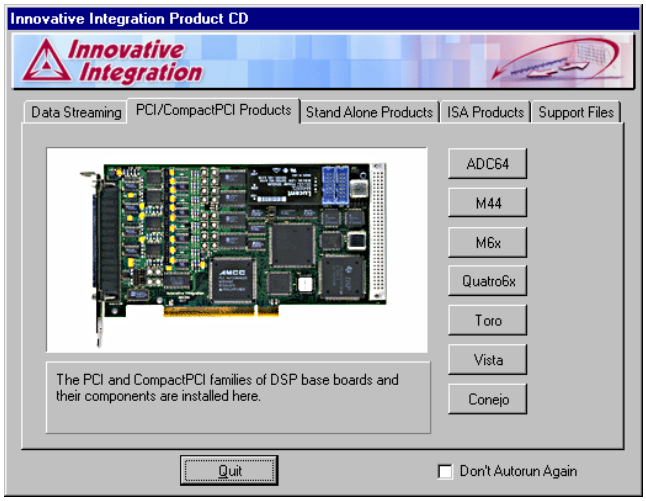
Software Installation

The development package has an installation program that will guide you through the installation.

Note: Before installing the host development libraries (VCL components or MFC classes), you must have Microsoft MSVC and/or Borland C++ Builder installed on your machine, depending on which of these IDEs you plan to use for Host development. If you are planning on using one of these environments, it is imperative that these environments be tested and known-operational before proceeding with the library installation. Additionally, you must install Code Composer Studio prior to installation of the development package software. If these items are not yet installed, then the installation program will not permit installation of the associated development libraries. However, drivers and DLLs-only may be installed, to facilitate field deployment.

Win2K/XP Users Only. You must have Administrator Privileges to install and run the software/hardware onto your system, refer to the Windows documentation for details on how to get these privileges.

To begin the installation, start the host operating system and insert the installation CD. The installation should autostart after you put it into the CD. If the CD does not auto start, click on the **Start** button, then **Run**. Enter the path to the SETUP.EXE program located at the root of your CD-ROM drive, i.e. E:\SETUP.EXE. The setup program will run. Select the appropriate tab. From there, select the appropriate baseboard button, depending on the type of hardware being installed. The baseboard-specific install screen will automatically come up. You should see a screen similar to the following. **Note:** If you



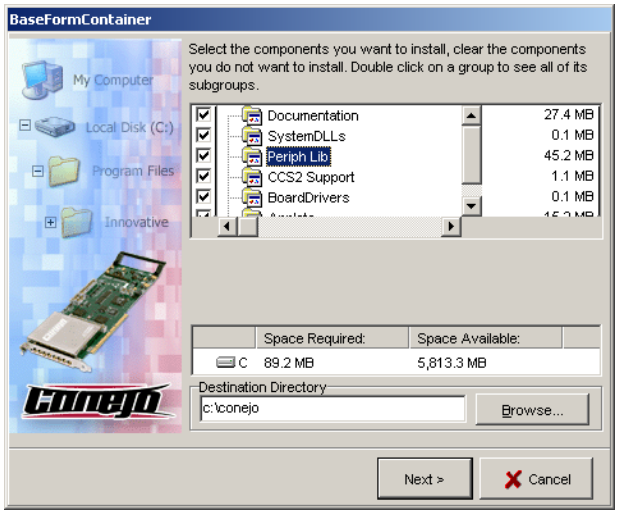
Conejo Installation
Innovative Integration
... real time solutions!
Welcome to the Conejo installation program. This program will install all the Conejo Armada Toolset components onto your computer.
Components to Install
Conejo - (including Applets, Examples and Drivers)
Armada - (including C++Builder or MS Visual C++ support)
Innovative - (including C++Builder or MS Visual C++ support)
Hasp Key Support - (software/hardware security key)
Event Log
Install
Abort
Exit

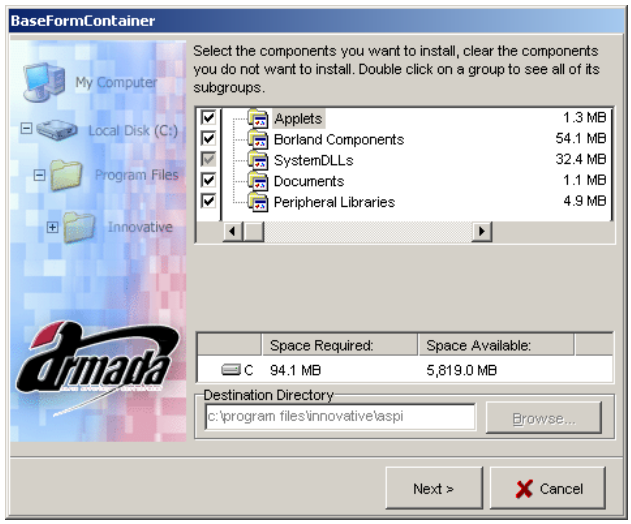
are attempting a full installation with development libraries for a DSP board and have not already installed Code Composer and your Host IDE (Borland Builder or MSVC), you should do so before proceeding with the installation and NOT click Install at the first selection window. In this case, click the

Exit button to terminate. If you are deploying a completed application and need driver and DLL support files only to be installed, proceed. There will be a chance to check drivers-only and DLLs-only in each of the sub-installs.

In the example above, each of the check boxes have sub-installs associated with them, and will open a sub-install screen window with another set of check boxes allowing you to select files to be installed. For example, the first sub-install for “Conejo - (including Applets, Examples and Drivers” is shown below.

The second image below, is for “Malibu - (including C++Builder or MS Visual C++ support)”. It follows a similar pattern and presents another set of choices as illustrated.



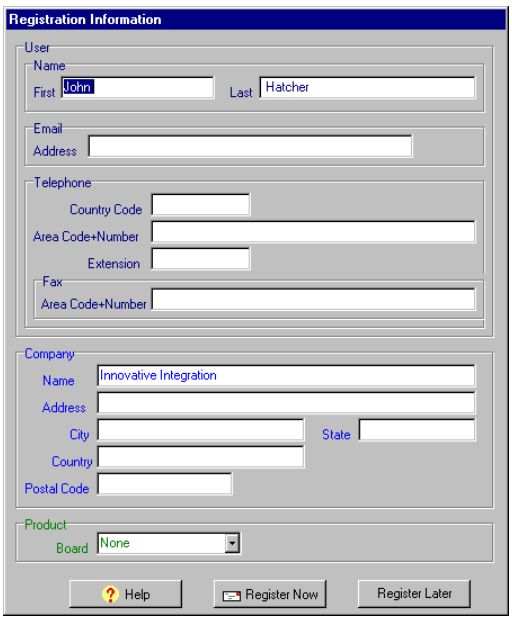


You may customize your installation, including or omitting components shown in these sub-install menus. At this point in each sub-install, check the desired components to be installed and click the ‘Next>’ button. Note that first-time development-system installs usually require the installation of all components.

The host examples included in this installation are written in C++ using Borland Builder or Microsoft MSVC, and C++ on the target side using Code Composer Studio. They are provided to illustrate the various features of the board and how these features are utilized.

Tools Registration

Before beginning DSP and Host software development, you must register your installtion with Innovative Integration. Technical support will not be provided until registration is successfully completed. Additionally, some of the development applets will not operate until unlocked with a passcode provided during the registration process. To Register, click Start | Program Files | M6713 | New User to start the *NewUser.exe* registration application. The registration form below will be displayed:

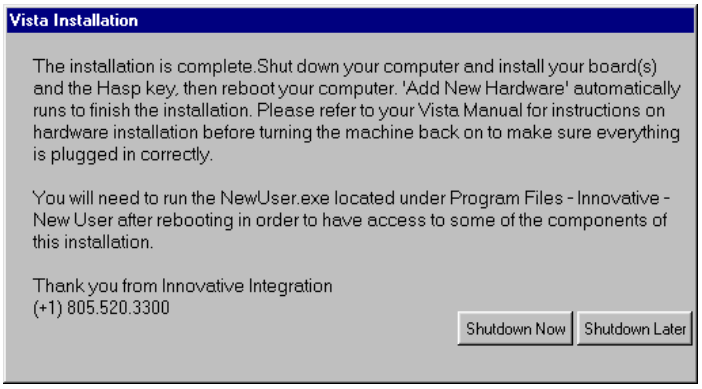


You should fill it out completely and return it to Innovative, either via email or fax. Upon receipt, Innovative will provide access codes to enable downloading of update software from our website, telephone hotline support and unrestricted applet access.

At this point you have finished the installation process and you are ready to begin development. Exit with the **OK** button.

Finally, the following screen appears. At this point, you should shut down your computer. This will also allow you to physically place the board(s) into the machine in addition to allowing newly installed components to be initialized

.


Vista Installation
The installation is complete.Shut down your computer and install your board(s) and the Hasp key, then reboot your computer. 'Add New Hardware' automatically runs to finish the installation. Please refer to your Vista Manual for instructions on hardware installation before turning the machine back on to make sure everything is plugged in correctly.
You will need to run the NewUser.exe located under Program Files - Innovative - New User after rebooting in order to have access to some of the components of this installation.
Thank you from Innovative Integration
(+1) 805.520.3300
Shutdown Now
Shutdown Later

Hardware Installation

The software components of the Development Package have been installed. To proceed with the Development Package Kit installation, it will be necessary to configure and install your hardware.

First, the emulator hardware must be configured and installed into your PC. The emulator hardware is described in the table below:

Type	**Features**
Pod-based	Uses a special ribbon cable with integrated line drivers to connect the target DSP emulation signals to the JTAG debugger card. Usable on 3.3 volt or 5 volt designs. (Including 'C54x and 'C6x)

PCI Pod-Based Emulator Installation

To install the PCI pod based emulator, follow the instructions below:

1. Shut down Windows and power-off the host system.
2. Perform the board installation in an "ESD" or static safe workstation.
3. Power off the host system and **touch the chassis** of the host computer system to dissipate any static charge.
4. Remove the card from its protective static-safe shipping container, being careful to handle the card only by the edges.
5. Touch the chassis of the PC to dissipate any built up static charge.
6. Securely install the JTAG board in an available PCI slot in the host computer.
7. Connect the JTAG pod to the host-pod cable. Connect the host-pod cable to the connector located on the end bracket of the JTAG PCI plug-in board.

DSP Board Installation

When installing the target card:

1. Power off the host system and **touch the chassis** of the host computer system to dissipate any static charge.
2. Remove the DSP card from its protective static-safe shipping container, being careful to handle the card only by the edges.
3. Install or place the M6713 card into an available PCI slot within your PC. If available, a 64-bit slot will provide optimal performance.
4. Connect the JTAG debugger pod cable from the JTAG board connection to the JTAG connector on the target board.
5. Connect the target cable between the JTAG PCI board within your PC to the mating JTAG pod connector.

A Few Considerations BEFORE Power-up. Double-check everything before applying power. Are the JTAG (if needed) and baseboard cards seated correctly in the slot?

It can't be overemphasized: double check your cabling BEFORE connection to the baseboard. Also, don't cut corners and hot plug the cables. This can cause latch-up of components on the card and damage them irreparably. Be aware that the cables to analog inputs are an important part of keeping the signals clean and noise-free. Shielded cables and differential inputs (where applicable) help to control and reduce the noise. After completing the hardware installation, boot up your PC.

After Power-up.

If using the Innovative Code Hammer debugger, Windows should detect and auto-configure the device at start-up. Under rare circumstances, Windows will fail to auto-install the device-drivers for the JTAG. If this happens, please refer to the "TroubleShooting" section.

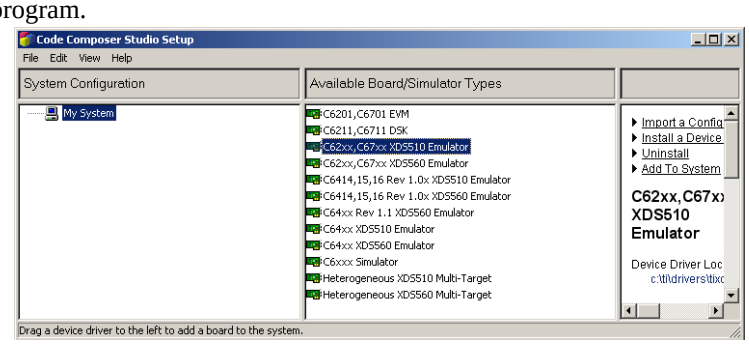
Use M6713LogiclLoader, to program the user logic. Please refer to Chapter # 8.

Before invoking the JTAG debugger, the user must boot the DSP using M6713Download utility. Please refer to Chapter # 8.

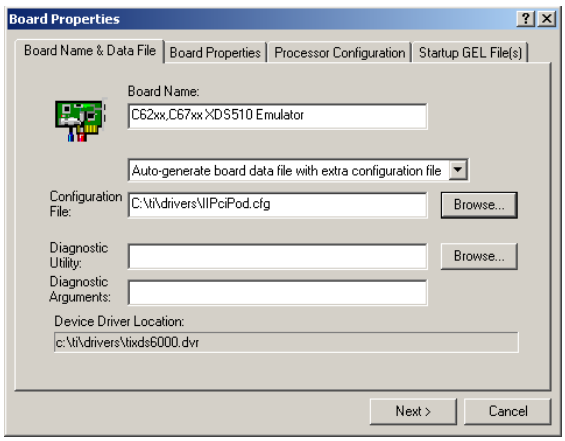
Note: The user must load the user logic and boot the DSP before start the Code Composer setup.

Code Composer Studio v2.x Setup

To setup Code Composer Studio v2.x and activate the Innovative-supplied Code Hammer JTAG board driver, the Code Composer Setup Utility must be run. Since the Code Hammer debugger is XDS510-compatible, Code Composer Studio setup must be configured to use the homogeneous XDS510 driver for the C6000. This driver is named C62xx,C67xx XDS510 Emulator within the Code Composer setup utility. In the figure below, it is the first item listed in the Available Board/Simulator Types column of the Setup program.

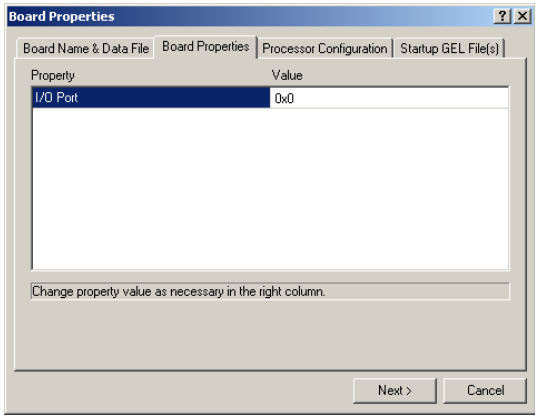


Click this driver from the “Available Board/Simulator Types” control within the setup utility and drag it into the “System Configuration” control. Then, right-click on this C6000 XDS object to invoke the Properties Dialog for the driver.

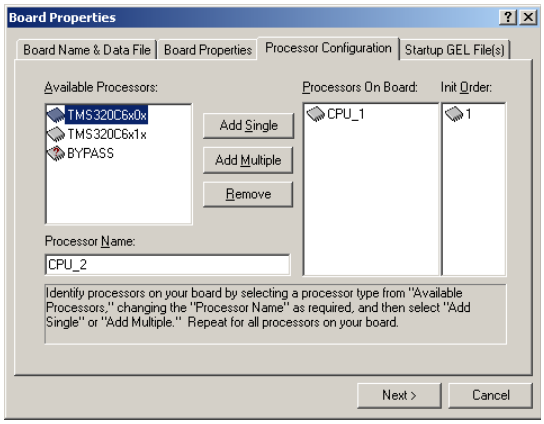


Under the “Board Name & Data File” tab, the board name edit box should list "C62xx,C67xx XDS510 Emulator". The “Configuration File” combo box should be changed to "Auto-generate board data file with extra configuration file". The “Configuration File” edit box should be changed to "<drive>:\ti\Drivers\IIPciPod.cfg", where <drive> is the letter for the drive onto which CCS 2.x was installed.

Under the “Board Properties” tab, the I/O port value for the driver should be set to virtual device address “0”.

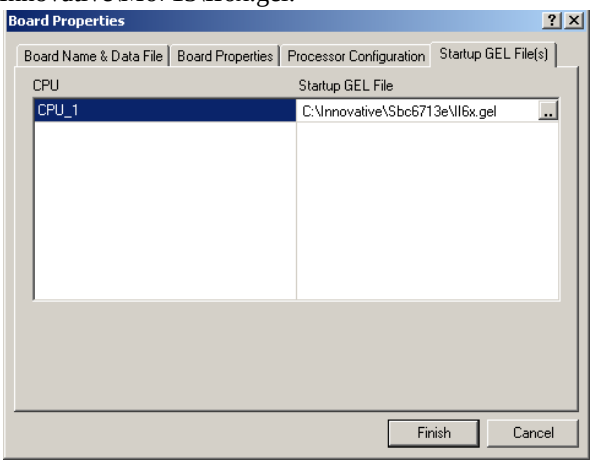


Under the “Processor Configuration” tab, processors of the appropriate type should be added to the “Processors on the Board” list box.



To configure for debugging the C6713 DSP, highlight theTMS320C6x1x, then click the Add Single button to add CPU_1 to the scan path as device 1 in the Init Order.

Under the Startup Gel Files tab, the Startup Gel File combo box should be the board-specific initialization GEL script for the target board. Innovative supplies an appropriate, board-specific GEL initialization file in the root of the board-specific libraries toolset directory. For example, for the M6713, this should be set to c:\Innovative\M6713\II6x.gel.

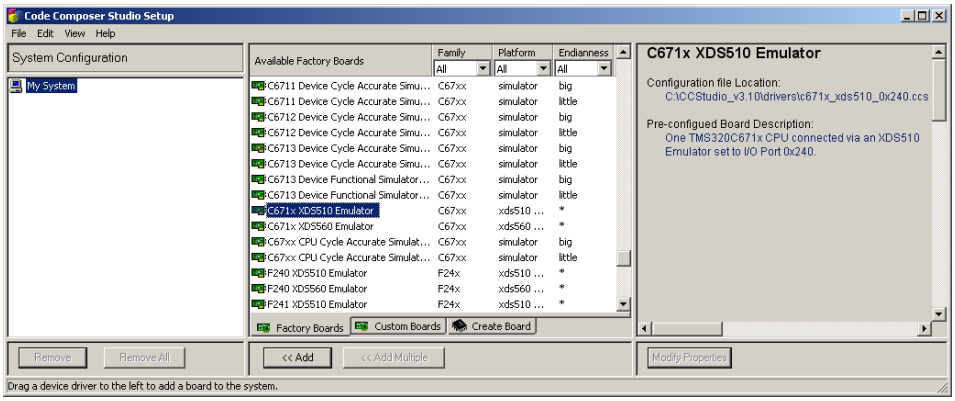


Finally, the setup configuration should be saved. After saving the configuration and shutting down the setup tool, Code Composer Studio should launch successfully. If you encounter difficulty launching CCS 2.x, run the JtagDiag.exe utility provided in your toolset to reset the debugger interface. Then, boot the 6713 DSP using M6713Coff Download utility. Then restart Code Composer Studio.

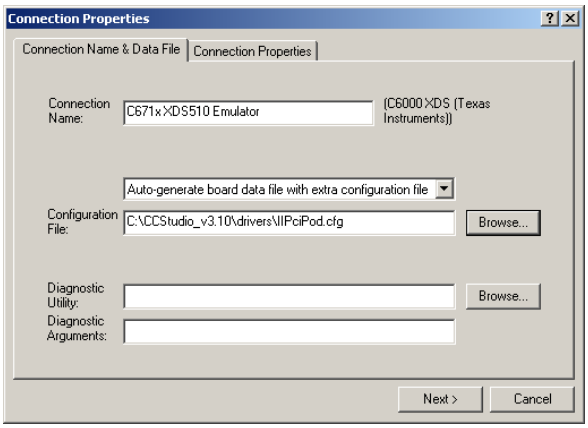
As a consequence of these steps, a new ccBrd0.dat file will be created. This file has been customized for the particular DSP target being used.

Code Composer Studio v3.x Setup

To setup Code Composer Studio v3.x and activate the Innovative-supplied Code Hammer JTAG board driver, the Code Composer Setup Utility must be run. Since the Code Hammer debugger is XDS510-compatible, Code Composer Studio setup must be configured to use the C671x XDS510 driver for the C6000. This driver is named “C671x XDS510 Emulator“ within the Code Composer setup utility. In the figure below, it is the second-to-last item listed in the Available Board/Simulator Types column of the Setup program.

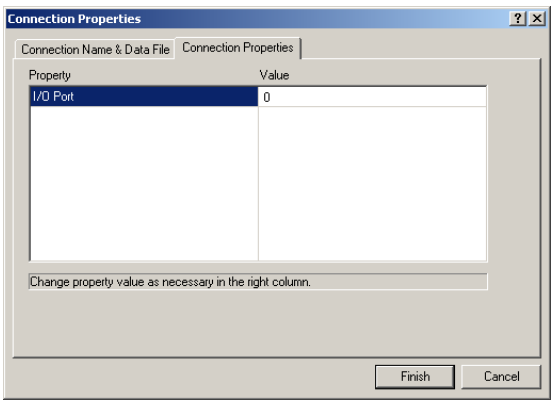


Click this driver from the “Available Factory Boards” control within the setup utility and drag it into the “System Configuration” control. Then, right-click on the newly-created “C671x XDS510 Emulator” board icon to invoke the Properties Dialog for the driver.



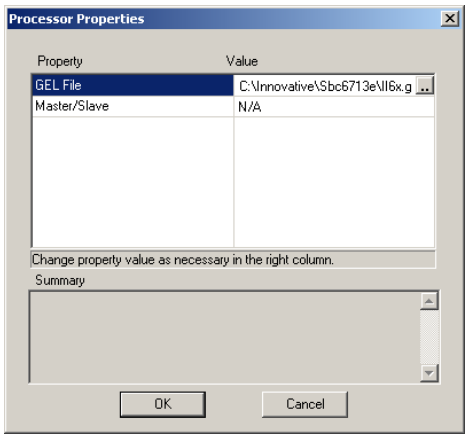
Within the “Connection Name and Data File” tab, the Connection Name edit box should list "C671x XDS510 Emulator". The “Configuration File” combo box should be changed to "Auto-generate board data file with extra configuration file". The “Configuration File” edit box should be changed to "<path-spec>\Drivers\IIPciPod.cfg", where <pathspec> is the valid path specification to the folder into which CCS 3.x was installed, including drive letter.

Under the “Connection Properties” tab, the I/O port value for the driver should be set to virtual device address “0”. Click Finish to dismiss this dialog.

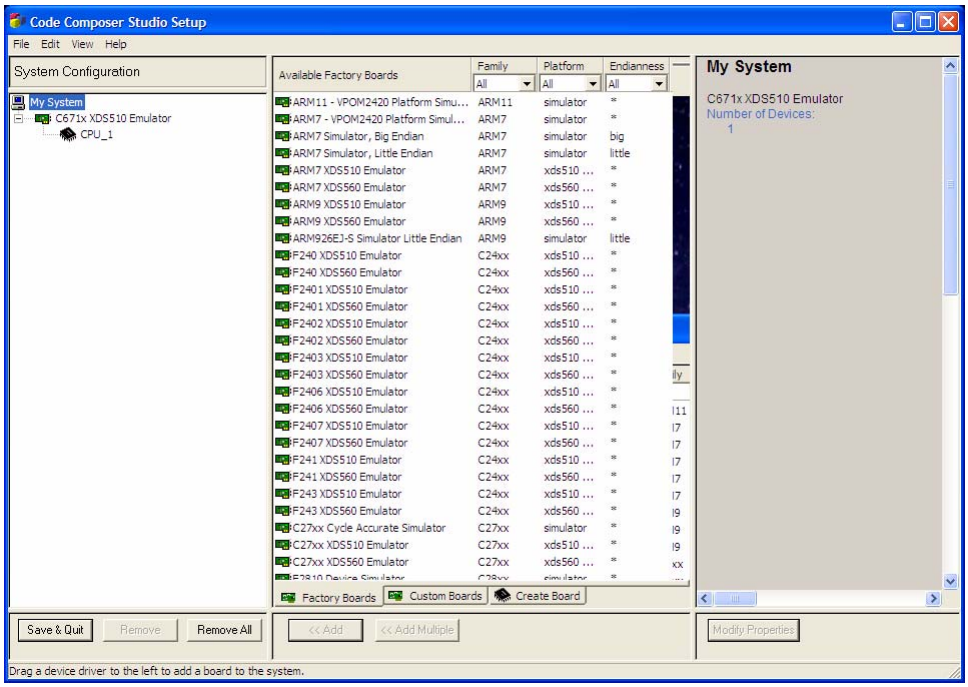


The M6713 has one DSP in the scan path ‘C6713 (a TMS320C6x1x device). Next, click on the “C671x XDS510 Emulator“ board icon within the System Configuration pane of the Code Composer Studio Setup window. The center pane will list all compatible processors which may be added to the JTAG scan path.

Next, right-click on the CPU_1 icon in the System Configuration pane, then click on Properties in the pop-up menu to invoke the Processor Properties dialog. Edit the Gel File value to point to the board-specific initialization GEL script for the M6713. The Innovative installation program automatically installs this file in the root of the board-specific libraries toolset directory. For example, for the M6713, this should be set to c:\Innovative\M6713\II6x.gel.



The resulting, final configuration is shown below:


Code Composer Studio Setup
File Edit View Help
System Configuration
My System
C671x XDS510 Emulator
CPU_1
Available Factory Boards
Family
Platform
Endianness
ARM11 - VPOM2420 Platform Simu... ARM11 simulator *
ARM7 - VPOM2420 Platform Simul... ARM7 simulator *
ARM7 Simulator, Big Endian ARM7 simulator big
ARM7 Simulator, Little Endian ARM7 simulator little
ARM7 XDS510 Emulator ARM7 xds510 ... *
ARM7 XDS560 Emulator ARM7 xds560 ... *
ARM9 XDS510 Emulator ARM9 xds510 ... *
ARM9 XDS560 Emulator ARM9 xds560 ... *
ARM926EJ-S Simulator Little Endian ARM9 simulator little
F240 XDS510 Emulator C24xx xds510 ... *
F240 XDS560 Emulator C24xx xds560 ... *
F2401 XDS510 Emulator C24xx xds510 ... *
F2401 XDS560 Emulator C24xx xds560 ... *
F2402 XDS510 Emulator C24xx xds510 ... *
F2402 XDS560 Emulator C24xx xds560 ... *
F2403 XDS510 Emulator C24xx xds510 ... *
F2403 XDS560 Emulator C24xx xds560 ... *
F2406 XDS510 Emulator C24xx xds510 ... *
F2406 XDS560 Emulator C24xx xds560 ... *
F2407 XDS510 Emulator C24xx xds510 ... *
F2407 XDS560 Emulator C24xx xds560 ... *
F241 XDS510 Emulator C24xx xds510 ... *
F241 XDS560 Emulator C24xx xds560 ... *
F243 XDS510 Emulator C24xx xds510 ... *
F243 XDS560 Emulator C24xx xds560 ... *
C27xx Cycle Accurate Simulator C27xx simulator *
C27xx XDS510 Emulator C27xx xds510 ... *
C27xx XDS560 Emulator C27xx xds560 ... *
Factory Boards
Custom Boards
Create Board
My System
C671x XDS510 Emulator
Number of Devices:
1
Save & Quit
Remove
Remove All
<< Add
<< Add Multiple
Modify Properties
Drag a device driver to the left to add a board to the system.

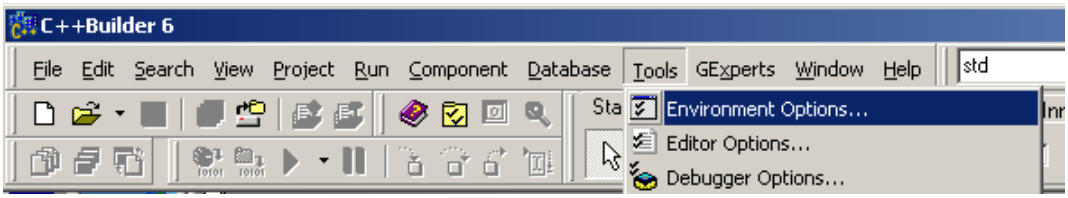
Finally, the setup configuration should be saved. After saving the configuration and shutting down the setup tool, Code Composer Studio should launch successfully. If you encounter difficulty launching CCS 3.x, run the JtagDiag.exe utility provided in your toolset to reset the debugger interface. Then restart Code Composer Studio.

As a consequence of these steps, a new ccBrd0.dat file will be created. This file has been customized for the particular DSP target being used.

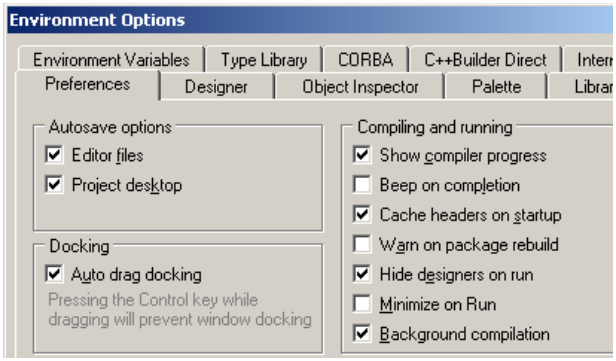
Borland Builder Setup and Use

Following the normal installation of the Innovative Integration toolset components, numerous VCL components and C++ classes are automatically added to the BCB IDE. Additionally, Innovative recommends that the following IDE and project options be manually changed in order to insure simplified use and proper operation:

Automatic saving of project files and forms during debugging. Invoke the Tools | Environment Options dialog from the main BCB toolbar.

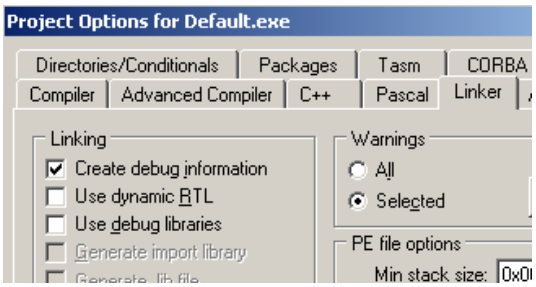


This will invoke the Environment Options dialog:

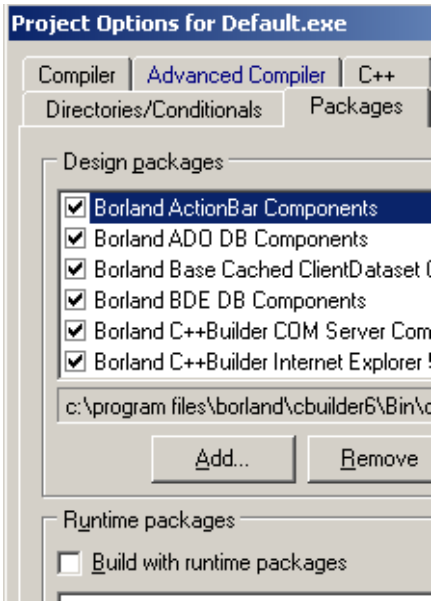


Enable autosaving of Editor Files and the Project Desktop, so that project files are automatically saved each time a project is rebuilt and debugged.

Static-binding of built executables. Click on Project | Options on the main BCB toolbar to invoke the Project Options dialog. Then click on the Linker tab. Uncheck Use Dynamic RTL checkbox.

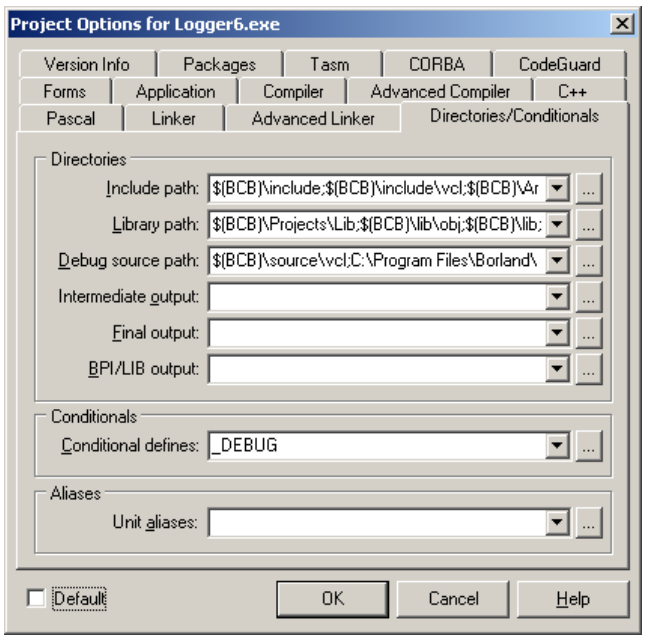


Next, click on the Packages tab and uncheck the Build with runtime packages checkbox.

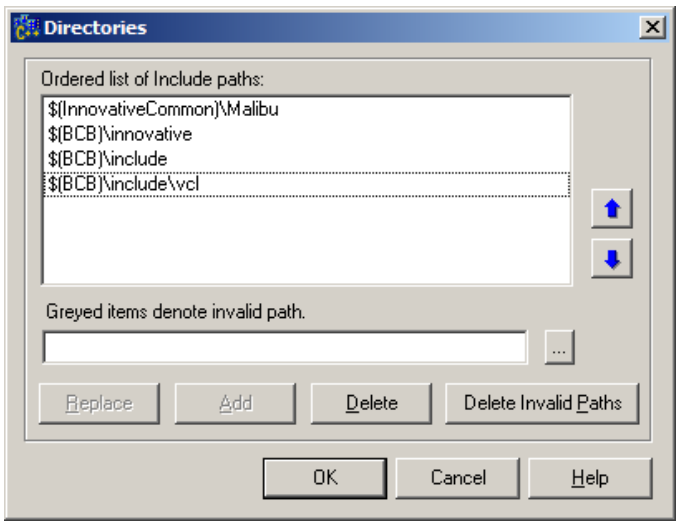


These options insure that projects are built with minimal dependencies on external DLLs. See the FAQ "**What DLLs do I have to deploy with my newly created executable"** in the Troubleshooting chapter for details on which DLLs must be deployed with user-written executables.

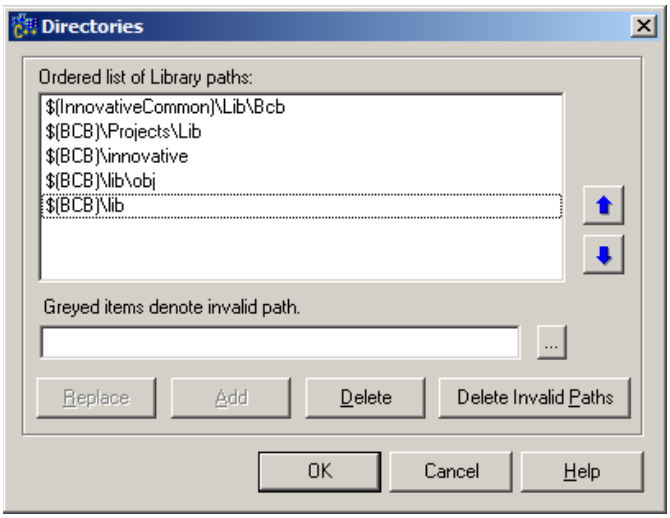
Appropriate library and include paths. Click on Project | Options on the main BCB toolbar to invoke the Project Options dialog. Then click on the Directories | Conditionals tab to edit the default Include and Library paths which should be used when constructing a Malibu-based application.



Next, click on the ellipses next to the Include Path edit box to invoke the Include Path editor dialog. Add an entry for $(BCB)\Innovative, then click Ok to accept these edits.



Next, click on the ellipses next to the Library Path edit box to invoke the Library Path editor dialog. Add entries as shown below, then click Ok to accept these edits.



These changes insure that the standard Malibu headers and object files are available to projects during compilation. Note that these paths may either be added to the default BCB project, by editing these options without first opening a specific project, or to specific projects after opening them. The advantage of the former is that the settings are automatically present on allsubsequently-created projects.

CHAPTER 3

About the Baseboard

DSP Baseboard Hardware Features

The M6713 baseboard features a TMS320C6713 digital signal processor with 128 Mbytes of SDRAM memory.

To complement this core, one or two modular Omnibus I/O modules may installed into the onboard I/O sites. A wide variety of I/O modules are available to address myriad application requirements. The combined baseboard/module system serves a variety of applications including servo applications, data acquisition, stimulus-response measurements and many others.

The tight coupling of the DSP, analog IO and other peripherals make the M6713 well-suited for a variety of application such as communications baseband processing, ultrasound applications, multi-axis controllers for high speed servos, RADAR, SONAR applications, communications signal processing and many data acquisition applications.

The baseboard has a variety of features that make it easy to develop high performance systems. On-card, very low noise power supplies provide clean power for analog peripherals installed into either Omnibus I/O site. Other features include 32 bits of programmable digital IO, FPDP data ports, advanced clocking mechanisms, multi-card triggering and clock sharing plus a flexible timebase for data sampling, and timer/counters. The M6713 features PCI bus-mastering interface which supports burst transfers are rates to 512 MB/sec to the host PC system.

Digital Signal Processor

The M6713 baseboard uses the TMS320C6713 DSP, operating at 300 MHz. This DSP is a 32-bit floating point device. The DSP interfaces to the memory and peripherals on the baseboard through its external memory interface (EMIF), which has programmable definitions for the memory interface timing.

DSP External Memory

The M6713 baseboard provides 128 Mbytes of SDRAM memory mapped to the '6713 DSP CE2 memory space. This memory provides space for program and data storage as well as storage space for data acquired/generated by the baseboard analog hardware. This memory is programmed to operate at 75 MHz regardless of the DSP core clock rate. The initialization of all external memory spaces are defined within the file `HdwLib\IIInit.cpp` and include the correct parameters for the type of SDRAM used on the baseboard, including refresh timing, as well as timings for all sync and async peripherals and should not be modified.

DSP Initialization

For proper operation of the external peripheral on the baseboard, the external memory interface control registers must be configured prior to use of the external memory interface. Applications built under the Pismo Toolset libraries will automatically initialize the registers appropriately (using code within `HdwLib\IIInit.cpp`). For those customers who need to initialize the registers manually, please refer to the EMIF register initialization values within the `IIInit.cpp` source file to obtain the required register values. Please note that the initialization is order sensitive and should be performed in the order given in the table below.

Register Name	Address	Value	Use
GBLCTL	0x01800000	0x00003078	
CE0CTL	0x01800008	0x21A28A22	FPDP
CE1CTL	0x01800004	0x0000C041	Asynchronous devices
CE2CTL	0x01800010	0x00000030	SDRAM
CE3CTL	0x01800014	0x11010420	Omnibus Modules
SDTIM	0x0180001C	0x00000350	
SDEXT	0x01800020	0x000544a7	
SDCTL	0x01800018	0x6B338000	

TABLE 2. 'C6713 DSP EMIF Control Register Initialization Values

During the development process, code may be downloaded to the baseboard using a JTAG debugger or via the PCI bus interface. After development is complete, the debugged application image may be downloaded from within a Host application using standard functions within the Malibu librares.

DSP JTAG Debugger Support

Standard TMS320 family JTAG debugger operation is supported by each baseboard. An external debugger connector is supplied that allows use of industry standard JTAG debugger hardware from Innovative, Texas Instruments, and other third party suppliers. The DSP is the only device in the scan path. Software for JTAG debugging and code development is TI Code Composer Studio.

The Pismo Class Library

Innovative Integration's ***Pismo*** is a software class library allows the developer to fully exploit the advanced hardware features of the Innovative DSP product lines and to reap all the benefits from Texas Instrument's DSP/BIOS Operating system. Every board peripheral has been carefully integrated into the OS and its functionality encapsulated in a device driver that can readily be controlled within DSP/BIOS applications including PCI interface, analog I/O, external bus and memory, serial ports and other I/O devices.

Pismo provides extensive C++ class support for:

- Dynamic creation and runtime control of tasks
- Simplified management of and access to all TI Chip Support Library (CSL) and DSP/BIOS API functions including: Semaphores, Mutexes, Mailboxes, Timers, Edma, Qdma, Atoms, McBsp, Timebases, Counters, etc.
- Foundation (base) classes for DMA-driven device driver development
- Templatized queues
- Partial standard-template library functionality via STLPort

For example, the code fragment below uses the Pismo `IntBuffer` class to initialize a QDMA (quick DMA) to perform a memory-to-memory move of a constant value (0) into a 4096-word buffer (at Src), then to copy the source buffer (Src) to the destination buffer (Dst):

```
// Create a source buffer of 0x1000 integers
IIBuffer Src(0x1000);
// Initialize the source buffer with zeros
Src.Set(0);
// Create a destination buffer of 0x1000 integers
IIBuffer Dst(0x1000);
Dst.Copy(Src);
```

Simple To Use. In the same way, peripheral-specific class libraries dramatically simplify access to board-specific peripheral features. For example, the code fragment below illustrates real-time processing and display of analog input signals running on the M6713 DSP board equipped with an Omnibus module within a separate thread of execution:

```
//---------------------------------------------------------------------------
//  LoopThread() --  Capture snapshots of A/D input
//---------------------------------------------------------------------------

class LoopThread : public Thread
{
public:
    LoopThread(IIPriority priority)
        : Thread(priority), FCount(0), Cursor(0), Requested(false)  { }

    int             Count()
        {  return FCount;  }
    void            Resize(int size)
        {  CaptureEvents=size;  Snaps.Resize(size);  }
    IntBuffer &     Acquire()
        {
        Cursor = 0;
```

```
        Requested = true;
        Available.Acquire();

        return Snaps;
        }

protected:
    // Fields
    volatile int    FCount;

    // Data
    bool            Requested;
    Semaphore       Available;
    IntBuffer       Snaps;
    int             Cursor;
    int             CaptureEvents;

    // Methods
    void Execute()
        {
        // echo input to output
        while(!Terminated())
            {
            AIn.Get();
            ++FCount;

            //
            // If main thread wants a block, copy it to him
            if (!Requested)
                continue;

            int Residual = Snaps.Ints()-Cursor;
            int Chunk = std::min(Residual, AIn.Buffer().Ints());
            Snaps.Copy(AIn.Buffer(), Cursor, Chunk);
            Cursor += Chunk;

            if (Cursor == CaptureEvents)
                {
                Requested = false;
                Available.Release();
                }
            }
        }
};
LoopThread Loop(tpHigher);
```

Not Just for C++ Experts . Note that even if you're not a C++ maven, the code is quite clear and understandable. In fact, one of the benefits of using C++ is that while it helps to mitigate and manage complexity to support creation of larger, more sophisticated applications, it is often simply used as a "better" dialect of the C language. C++ is essentially a superset of C. As such, you may freely intermix calls to legacy 'C' functions, newly-written C functions, Assembler functions and C++ functions (called methods) within C++ programs. You need not fully understand all of the enhanced capabilities and features of C++ in order to fully exploit the features of the class libraries provided in Pismo.

Analog I/O Streams

The Analog I/O is, for most applications, the most important feature of the M6713 baseboard. Most of the peripherals on the hardware are related to Analog I/O. Most of the configuration options are related to Analog I/O. It is the part that causes the most problems in development.

To maximize the chances for success, the Pismo library provides a set of classes that hide all of the details of data acquisition. From the application level, the user simply processes buffers of data. The details of hardware and software management are isolated from the application.

Stream Objects and Device Drivers

Data I/O in DSP/BIOS is accessed and controlled via custom device drivers. Access to the device driver is controlled by a Stream class. These drivers are dynamically installed by the Stream when needed by the user application. From the point of view of the application the stream control class provides all of the user interface function needed to configure and operate the I/O operation.

TABLE 3. Device Driver and Stream Classes

Device Driver Class	Stream Class	Description
AnalogInputDriver	AnalogInStream	Streamed Input from an analog source. Continuous data flow with buffering. Data flow stops only via trigger control.
AnalogOutputDriver	AnalogOutStream	Streamed Output to an analog source. Continuous data flow with buffering. Data flow stops only via trigger control.
CaptureInputDriver	CaptureInStream	Burst Input from an analog source. Data flow is discontinuous, filling each buffer requested and stopping.
ServoBase	ServoIntf	Continuous, low-latency analog capture and playback suitable for performance servo control applications. Event-at-a-time application data processing.
Stream	FpdpInStream	Burst Input from the FPDP hardware.
Stream	FpdpOutStream	Burst Output to the FPDP hardware.

Hardware Isolation and Independence. The Analog and the Capture driver allow a single Stream to be used with different analog hardware. In the M6713, for example, Omnibus modules allow a wide variety of analog choices on a single baseboard. Each of the Stream classes can be “attached” to a particular module and will automatically configure itself to use that hardware.

The Servo driver provides low-latency, interrupt-driven data processing, suitable for real-time control applications, albeit at the expense of high CPU usage. It is currently implemented only for the Servo16 module.

The FpdpInStream and FpdpOutStream drivers provide communications with external Front Panel Data Port devices.

Stream I/O Types. There are two distinct categories of Streams implemented within Pismo - Continuous and Burst.

Continuous Streams use the model that the input or output is a continual process, whether periodic or not. Thus in order to avoid data loss when the application is momentarily busy, internal buffering is provided so that the hardware may operate for extended periods without software intervention. This means latency must be increased: data may be 'in the queue' for some time until additional data forces it out to the application. Of course, if the application does not process the data as fast as it arrives data will eventually be lost.

Burst Streams use a different model. Here data movement is 'on demand' instead of asynchronous. If no request for action by the application is received, the Stream is idle. This type of Stream is more common for non-Analog I/O such as the FifoPort or PCI busmastering, but the CaptureInStream implements a burst type I/O model on the analog hardware.

Stream Buffer Model. Each Stream uses data buffer class objects to pass data between its hardware and the application. These buffers are all the same size. Passing data between Streams is simple if the buffers are chosen to be the same size:

```
Ain.Get();
Aout.Put(Ain.Buffer());
```

Buffer transfers are efficient because the data buffers are not copied at any time during the transfer process. By default, Streams allocates three buffers, two internal and one swap buffer. If desired, the number of internal buffers in the pool may be modified prior to opening the Stream by assigning a new value using the `BufferCount` method.

```
// Instantiate the analog stream objects
AnalogOutStream Aout;
Aout.BufferCount(5);
AnalogInStream Ain;
Ain.BufferCount(5);
```

This code forces the Stream to allocate five internal buffers.

The size of data buffers may be specified explicitly using the `Stream::Events` method. This latter method sizes the buffers such that they can contain the at least the specified number of acquisition "events", where an event is defined as one sample from all enabled A/D or D/A channels. This simplifies most buffer processing algorithms since all buffers are guaranteed to contain an integral number of samples from all enabled channels.

The product of the buffer size and the number of buffers gives the load-carrying capacity of the system. For example, the originally allocated three buffers per stream, each sized at 0x1000 bytes, running at 44.1 kHz equates to a load carrying capacity of

```
(0x1000 bytes/buffer) x (3 buffers) / (44100 samples/sec) /
(2 bytes/sample) = 139 mS
```

Whereas in the second example, with six buffers per driver pool

```
(0x1000 bytes/buffer) x (6 buffers) / (44100 samples/sec) /
(2 bytes/sample) = 278 mS
```

Data integrity can thus be preserved at the expense of additional memory utilization.

Burst Streams place data into the buffer provided. No buffering is used, and data acquisition is halted when the provided data buffer is filled.

Stream Internals. The DSP CPU used on the M6713 is powerful and fast, yet the Stream classes improve performance even further by drastically reducing CPU use for data movement. The available DMA channels in the C6000 DSPs are fully exploited to do movement to and from hardware to memory so that hardware interrupt rates rarely exceed 1KHz! The net effect is that virtually all of the bandwidth of the CPU is available for application processing, without requiring any application DMA programming.

Multitasking Friendly. The Stream classes support efficient cooperation in multitasking applications. Any function that requires a delay to complete. will block using DSP/BIOS functions that release other OS threads for efficient utilization of the processor.

Using Analog Streams in an Application

The AnalogInStream, AnalogOutStream, and CaptureInStream all allow fast data movement between the application and the hardware in different modes. Once associated with a hardware device, they allow all configuration and control of the session to take place through the methods of the Stream.

Every Stream must consider these questions to make a functioning application:

- Which stream to use (Input vs. Output or Continuous vs. Burst).
- Which hardware to use, and in which hardware mode.
- Which clock source to use, and with what parameters.
- Which triggering mode to use, and with what parameters.

Once these are taken care of, using the Stream to perform the Analog I/O is a simple matter. Consider the code fragment below which illustrates all of the steps necessary to fully initialize and stream a stream a continuous 1 kHz sine wave to the analog outputs present on a SD16 Omnibus module attached to an M6713 baseboard SD16 module at 50 KHz:

```
using namespace II;

//---------------------------------------------------------------------------
//  IIMain() --  Illustrate Omnibus Output
//---------------------------------------------------------------------------

AnalogOutStream  Aout(Omnibus::mSite0);

void IIMain()
{
// ...Load Module onto site 0.
LoadModule(Omnibus::mSite0, Omnibus::mtSD16);
//
//  ...Use default clock (DDS)
//  ...Set Clock Rate
ClockRateUIPtr(Aout.Clock())->Rate(50000);
//
//  ...Output on all channels
Aout.Channels()->EnableAll();
//...Size the buffers
Aout.Events(5000);
Aout.BufferCount(BuffersPerSec/3);

// Open the streams
```

```
Aout.Open();

// Stream loop...
bool run = true;
while ( run )
{
Aout.Generate();
Aout.Put();
}

// Terminate streaming
Aout.Close();
}
```

Examining the above code, you can see the application uses an AnalogOutStream, since it is an output program. The next interesting line is the call to LoadModule(). This informs the system of which module is 'plugged in' on Omnibus::mSite0, where the stream is also attached. With the attachment of the module to the stream, the stream object can configure hardware, clock, and trigger settings.

The next several lines configure the Stream clock rate, the channel configuration, and the stream buffer count and buffer size. Then comes the `Stream::Open` method which activates the device driver in DSP/BIOS. Afterwards, the `Stream::Control` method may be used to perform any necessary device-specific initialization and/or control functions.

Data flow begins with the calls to `Generate()` and `Put()`. `Generate()` uses signal generator classes to write a signal pattern into the buffer. `Put()` enqueues the data for output. Data output will not actually begin until the buffer queue is essentially filled with data. This avoids under-runs of the output. For input Streams, the `Get()` method starts streaming at the first call. These buffer methods should be repeated to keep the streams flowing. After use of a device is complete, it is closed using the `Stream::Close()` method.

Note that in the above example, the type of module used matters very little in the finished application. Simply by changing a single constant this code can be rebuilt to work on any module that supports Analog output. The module specific details are handled by the Pismo library internally.

Selecting the Stream Object. Each Stream object is used to manage input or output on a single Omnibus module. Multiple module applications need to use separate instances of the stream for each module site. The Stream object is associated with a module site by the constructor, allowing access to the hardware for configuration:

```
AnalogInStream  AIn(Omnibus::mSite0);
```

The `AnalogInStream` provides continuous streaming input. All data is delivered to the ring of internal buffers and from there to the application.

`AnalogOutStream` allows continuous streaming output to an output device. The application must deliver data as fast as it is consumed to avoid buffer underruns.

The `CaptureInStream` is a new type of driver that is used to emulate manual capturing of data to a buffer. In this mode, the analog hardware is inactive until a buffer is presented for filling. When this occurs, an acquisition is started that will fill the presented buffer, after which data taking is stopped. The process repeats for each buffer presented.

In the capture mode, no processing is taking place when data is not requested. This is a major difference from `AnalogInStream`. Also, if the buffer is sized to be smaller than the analog hardware's own FIFO or

storage, each buffer will be a snapshot dump of the FIFO contents. This makes capture useful for snapshots of very high rate analog input, faster than the module can be read. There is no way to take continuous data sets larger than a single buffer in capture mode. There will be a gap between any two captures. `AnalogInStream` should be used for continuous applications.

Selecting and Configuring Hardware. The M6713 supports Omnibus Modules, allowing multiple hardware configurations on the same baseboard. Modules are attached to an Omnibus site by a call to LoadModule().

```
// ...Load Module onto site 0.
LoadModule(Omnibus::mSite0, Omnibus::mtSD16);
```

Once a module is loaded, it can be accessed by the stream object's Module() method. This returns a generic pointer that can be converted to an exact module pointer to allow its methods to be called. The functions that do this are called Module Conversion Functions.

Module Conversion Functions are provided for all supported Omnibus modules. See the online help for the module class for a description of these functions. An example of the use of these functions:

```
SD16 * sd16 = SD16Ptr(AIn.Module());  // Gives valid SD16 object
A4D4 * a4d4 = A4D4Ptr(AIn.Module());  // Returns 0 -- not an A4D4!
```

Note that a module conversion function will fail if the conversion can not take place.

Selecting and Configuring Clocks. On attachment of the stream to a module, the Clock configuration system becomes active. It consists of the following methods:

TABLE 4. Stream object Clock Methods

Method	Description
IsClockSourceSupported()	Returns True if a clock source is allowed on the hardware.
SetClockSource()	Change clock source to the selected source.
Clock()	Returns an interface object that allows the configuration of the Clock source.

The User Interface object returned by the Clock() method is used to configure the selected clock source. The possible ways a clock can be configured depends strongly on the type of source. For example, an internal clock such as the DDS can have the clock frequency programmed. An external clock can not. Rather than provide a complicated set of functions, many of which may not work for a clock source, we instead separate each distinct part of the User Interface into separate interface classes. The interface object for a clock source may support none, or one, or any number of all the possible UI interface classes. An interface can be accessed by the conversion function for each of the UI interface classes. If an interface is not supported, the conversion function returns a null pointer.

The following code sample shows the use of conversion functions and multiple interface classes. The DDS clock source is in use. It supports both the ClockRateUI interface, which allows changing the clock frequency, and the ClockSyncUI interface, which allows configuration of the SyncLink/ClockLink master hardware to drive the DDS clock signal off the baseboard for use as a source on another board.

```
//  ...Set Clock Rate (allowed on DDS)
```

```
ClockRateUIPtr(AIn.Clock())->Rate(50000);
ClcokSyncUIPtr(AIn.Clock())->SyncLinkChannel(scSyncLink0);
```

Each line can be read from the inside out. `AIn.Clock()` returns the current clock UI object. This object is input into the clock conversion function `ClockRateUIPtr()` and converted into the `ClockRateUI` interface. Finally, the `Rate()` method of this class is called to set the rate of the clock to 50,000 Hz.

Selecting and Configuring Triggers. The Analog Stream objects allow the user to configure the triggering method used during the run. Triggering features are divided into four parts:

- Pretriggering - Handling data before the start trigger.
- Start Trigger - How to start data taking.
- Stop Trigger - How to stop data taking.
- Retriggering - How to handle interval before next start trigger

Selecting Pretriggering Modes. The pretrigger control for a stream consists of the following methods:

TABLE 5. Stream object Pretrigger Methods

Method	Description
IsPretriggerTypeSupported()	Returns True if the pretrigger mode is allowed on the hardware.
SetPretriggerType()	Change pretrigger to the selected mode.
Pretrigger()	Returns the pretrigger interface for the mode.

:

TABLE 6. Stream object Start Trigger Methods

Method	Description
IsStartTriggerTypeSupported()	Returns True if the start trigger mode is allowed on the hardware.
SetStartTriggerType()	Change start trigger to the selected mode.
StartTrigger()	Returns the start trigger interface for the mode.

:

TABLE 7. Stream object Stop Trigger Methods

Method	Description
IsStopTriggerTypeSupported()	Returns True if the stop trigger mode is allowed on the hardware.
SetStopTriggerType()	Change stop trigger to the selected mode.
StopTrigger()	Returns the stop trigger interface for the mode.

:

TABLE 8. Stream object Retrigger Methods

Method	Description
IsRetriggerTypeSupported()	Returns True if the retrigger mode is allowed on the hardware.
SetRetriggerType()	Change retrigger to the selected mode.
Retrigger()	Returns the retrigger interface for the mode.

Trigger configuration presents the same problem as the clock configuration, except more so. Triggering consists of four parts, each of which can be independently set. In addition, there are far more ways of defining triggers than there are for defining parts of a timer. For example, the start of data flow on Omnibus modules can be based off of an external digital signal, the value of the data on an input channel, by software command, or be automatic. Pretrigger and Stop trigger options are also numerous. A class that had methods for all these features would be large, complex, and would usually have most of its functions inoperative without giving the application any feedback.

Trigger configuration is also complicated by Module differences. Even the same type of trigger can differ in on different modules. For example, an external start trigger may allow the condition of the input signal (edge or level triggering, positive or negative polarity) be changed. Another module might not support changing these features, being always positive edge triggered. Another example is that threshold triggering might be able to be triggered of any selected channel, or it might be restricted to a predetermined channel.

The triggering configuration system uses trigger interface objects to allow configuration of the triggering modes. There are four access functions (Pretrigger(), StartTrigger(), StopTrigger(), and Retrigger()) giving interface objects for use. Each of these objects supports none, or one, or any number of Trigger UI interface classes for configuration, each of which can be exposed by a conversion function for the class. If a UI interface is not supported, the conversion function returns a null pointer.

The following code sample shows the use of trigger conversion functions and multiple interface classes. The AD40 module is in use. It supports several different Pretrigger and StartTrigger modes. In the example we wish to use Counted Pretriggering and Threshold Start Triggering.

```
//
//  This example for an AD40 uses pretriggering and
//     start triggering...
//
//  Set triggers
Stream->SetPretriggerType(TriggerManager::ptCounted);
Stream->SetStartTriggerType(TriggerManager::stThreshold);
//
//  Configure Pretrigger to 500 counts
CountedPretriggerPtr(Stream->Pretrigger())->PretriggerCounts(500);
//
//  Set start threshold to .1 volts
VoltageThresholdPtr(Stream->StartTrigger())->ThresholdLevel(.5);
ConfigurableTriggerPtr(Stream->StartTrigger())->Type(ttEdge);
ConfigurableTriggerPtr(Stream->StartTrigger())->Polarity(tpPositive);
```

Counted Pretriggering allows the preservation of data samples before the start trigger fires. Normally the first data point read was taken just after the start trigger fires. With Counted Pretriggering, N samples *before* the trigger fires are output when the trigger fires. In the example below the pretrigger is set to return 500 samples from before the trigger.

The AD40 Threshold mode supports two interfaces: VoltageThreshold and ConfigurableTrigger. VoltageThreshold configuration allows the threshold to be set in volts with the `ThresholdLevel()` method. In the above example the hardware is configured to trigger at half a volt. The ConfigurableTrigger UI interface allows signals to be specified as Type edge or level, and Polarity positive or negative. In the example above, the trigger is configured for a positive-going edge. This means that when the signal crosses the threshold from below to above, the start trigger will fire. A crossing from above the threshold to below it will not fire the trigger.

Each line can be read from the inside out. `Stream->StartTrigger()` returns the current start trigger UI object. This object is input into the trigger conversion function `ConfigurableTriggerPtr()` and converted into the `ConfigurableTrigger` interface. Finally, the `Type()` method of this class is called to set the trigger type to `ttEdge`.

The trigger modes a module supports and the UI interfaces its supported modules support are very module dependent. It is quite common to have to use several conversion functions to configure a trigger mode. It is also common for a trigger to be unconfigurable, exposing no trigger UI classes. Similarly, many modules support several triggering modes. Other modules support only the default, unconfigurable combination of no Pretriggering, Always start, Never stop, and no Retriggering. The description of the modes a module supports and the UI interfaces a module supports in each mode are listed in the online help with the description of each module.

Interrupt Handling

In DSP/BIOS, all hardware interrupts are intended to be managed by a DSP/BIOS hardware manager. This manager allows user functions to be called as part of the interrupt process while still cooperating with DSP/BIOS. As a part of the configuration process, the user can direct the HWI manager to call a user function.

Interrupts in a C++ Environment. In a system using C++, this means of attaching interrupts leads to several difficulties. A minor problem is that of name-mangling. C++ creates a new name for every function created in order to allow overloaded functions. The DSP/BIOS configuration does not understand the new name and results in a linker error. There is a simple work-around for this:

```
extern "C"
{
    void  MyHandlerFunction( void * arg );
}
```

This declares to the compiler to create a standard C symbol name for this function (`_MyHandlerFunction`) which can be used by to the DSP/BIOS configuration tool.

A more fundamental problem is that this mechanism does not allow the interrupt handling function to be changed during the life of the program. Also, this handler function may not be a class member function. This restriction can make designing a class object that handles interrupts awkward.

The Pismo Solution. The solution implemented in the Pismo environment is to take over all interrupt handling by providing a full set of standard handlers. The user then never needs to work in the CDB editor to provide handlers. The standard Pismo handlers contain code that will call a user's installed interrupt handler function if one is provided. While this adds a small amount of latency to the interrupt, the DSP/BIOS overhead per interrupt call is still much greater and dominates the total time per interrupt.. In

general, the BIOS environment is not suited for extremely high interrupt rates. Luckily, the use of DMA to aquire data from FIFOs on peripherals means that high rate interrupt handlers are not needed.

Pismo uses a special object, a Binder, to group a handler function and its arguments in a way that can be properly called by the standard handler. One form of Binder is used to attach a stand-alone function and its arguments, another form allows the binding of an Object, a member function of that object, and its arguments. This form of binder can allow a class object instance variable to act as a handler for interrupts. Here is an example from the **Messages** example of defining a binder for a timer interrupt:

```
//
//  Timer Interrupt Handler Function
void  OnTimerFired(int arg);
//
//  Binder Object for Timer
typedef void (*IntFtnType)( int arg );

FunctionHandler<IntFtnType, int> TimerBinder(OnTimerFired, 0);
```

And attaching the binder to an interrupt:

```
//  Set up a real time clock to send commands to host on
//    Target channel...
Irq Timer0( intTimer0 );
Timer0.Install( TimerBinder );
Timer0.Enable( false );
//
//  Turn on the clock at 5 hz
DspClock  Tclk0(50.0, 150.0);

Timer0.Enable( true );
```

In the example, `TimerBinder` is an object that collects the handler function, `OnTimerFired`, and its argument, 0. This object is passed into an `Irq` object associated with the `TCLK0` interrupt. When the timer interrupt fires, the handler will be called with its argument. The binder is a template, allowing any type of argument to be used with an interrupt handler.

Class Irq

Class `Irq` is an object that can be created to manage a specific interrupt. It has functions to set, clear, enable and disable the interrupt and also allows a handler to be installed that will be called whenever the interrupt fires. In the above code, see how all functions involving the interrupt were encapsulated in the methods of the `Timer0` class object.

Interrupt Lock Classes. A common need in a program is the ability to disable a particular interrupt, or all interrupts, in a portion of the program. The standard means of standalone functions (an disable followed by a enable interrupts) has a few problems. The first is that the means does not nest well. If a function blocking interrupts is nested in a second one, interrupts will be re-enabled at the wrong time. A second is that if the function has multiple return paths, each must have the re-enable code in it. The introduction of C++ exceptions makes this problem even worse.

The Pismo library provides a set of class objects that meet this problem. These lock objects disable a particular interrupt or all interrupts in a region and restore the state to what it was on entry when the

lock object is destroyed. If the object is created on the stack, any means of exiting the block in which the object is defined will cause the cleanup code to be called. Calls to these objects properly nest as well.

TABLE 9. Interrupt Lock Classes

Lock Class	Interrupts Affected	TI Class Library
`InterruptLock`	One IRQ	CSL.
`GlobalIntLock`	All interrupts	CSL.
`HwiGlobalIntLock`	All interrupts	DSP/BIOS.

Interrupt Binder Templates

The Binder system can be thought of as a more flexible and powerful version of a function pointer variable, allowing a user callback function to be called indirectly without knowing more than the interface to the function. Since the binder objects are templates, the type of the function and its arguments are not fixed but can be of any type. Also, member functions can be bound to an interrupt, which a callback function can never do.

The Binder system is powerful, yet in practice is quite simple to use. This system illustrates the power of the C++ language to contain a complicated system in a simple-to-use package.

Class InterruptHandler. This class is a base class for the `ClassMemberHandler` and `FunctionHandler` templates. It provides the interface the Pismo system uses to call the interrupt handler.

Class ClassMemberHandler Template. This template allows the binding of a member function of a class object with the object to call and an argument of any type. In this example the `IsrHandler` class is bound to a timer interrupt:

```
class IsrHandler
{
public:
    IsrHandler()
        : Binder(*this, &IsrHandler::MyHandler, &Tally), Tally(0)  { }
    ClassMemberHandler<IsrHandler, unsigned int *> Binder;

    void MyHandler(unsigned int * tally)
    {
        *tally += 1;

        if ((*tally & 0x7f) == 0)
            rtdx << "Isr tally: " << *tally << endl;
    }


private:
    // Data
    unsigned int    Tally;
};

// Instantiate a concrete instance of above class..
IsrHandler  Isr;

void IIMain()
{
    // Dynamically create an Irq object tripped from onchip timer 0
    Irq Timer0( intTimer0 );
    // Bind and install the interrupt vector
```

```
    Timer0.Install( Isr.Binder );
    // Program onchip timer 0 to signal at 100 Hz
    Timer0.Enable( false );
    DspClock Clock(100, 150, true, 0);
    Timer0.Enable( true );

    // Use RTDX event log to monitor progress
    rtdx.Enabled(true);

    rtdx << "Message from within IIMain,,,"<< endl;

    // Go to sleep...
    while (1)
        TSK_yield();
}
```

In the above example, the handler uses a **int *** argument to pass out information from the interrupt routine.

Class FunctionHandler Template. This template allows the binding of stand-alone function with an argument of any type. In this example the `OnTimerFired` function is bound to a timer interrupt:

```
//
//  Timer Interrupt Handler Function
void  OnTimerFired(int arg);
//
//  Binder Object for Timer
typedef void (*IntFtnType)( int arg );

FunctionHandler<IntFtnType, int> TimerBinder(OnTimerFired, 0);
```

This is the installation of the handler in the program:

```
    //  Set up a real time clock to send commands to host on
    //    Target channel...
    Irq Timer0( intTimer0 );
    Timer0.Install( TimerBinder );
    Timer0.Enable( false );
    //
    //  Turn on the clock at 5 hz
    DspClock  Tclk0(50.0, 150.0);

    Timer0.Enable( true );
```

EDMA and QDMA Handling

The TI C6000 processor supports a rich, powerful DMA engine to move data without CPU intervention. There are two kinds of DMA allowed. One, EDMA is full featured but can take some time to set up. QDMA is TI's facility for quick DMA movement of data. It is similar to a normal DMA transfer except that it is software triggered and performs only a single transfer. No linking of blocks is permitted with QDMA. It also is faster to initiate as only a few registers need to be set to start a new transfer.

Both kinds of DMA use a set of registers to define the configuration of a DMA transfer. By properly configuring the settings, many different transfer types can be performed, such as interelaved data, two dimensional arrays, and so on. See the TI Peripheral Library guide for more information on configuring EDMA and QDMA.

The QDMA has a single set of configuration registers, so only one QDMA may be in progress at the same time. The EDMA has a pool of blocks that may be used to define simultaneous, complex transfers.

Class DmaSettings. The DmaSettings class manages an image of the settings registers used to configure a QDMA or EDMA transfer. It provides properties to read and set the individual fields of the registers, saving the user the effort of masking bits and shifting data. It even provides functions that preconfigure some commonly used transfers, saving even more programmer effort.

The following code fragment shows how the setter functions are used to set up for a transfer. The DmaSettings class returns a reference to self on all setter functions, allowing multiple parameters to be set on a single line:

```
DmaSettings Cfg;

Cfg.Priority(DmaSettings::priHigh).ElementSize(DmaSettings::is32bit)
Cfg.SourceIncr(DmaSettings::Incr).DestinationIncr(DmaSettings::Incr);
Cfg.TCInt(true).TCCode(1).FrameSync(true);
Cfg.SourceAddr((int)&src_array[0]).DestinationAddr((int)(dest_array+50));
Cfg.ElementCount(50).ElementIndex(1);
Cfg.FrameCount(0).FrameIndex(1);
```

Class Qdma. This class manages the posting of Qdma requests. It contains functions to allow configration of a transfer, initiating a transfer and completion notification via either an interrupt or a polling function. Because the system state is saved in the object, transfers can be predefined and saved to be posted at a later time.

As with all DMA objects, the Qdma object uses an internal DmaSettings object to define the transfer. The Settings() method provdes access to the object to allow calling the DmaSettings classes own configuration functions, or configurations can be loaded from a second object with the Load() method.

```
// Q is a Qdma object, here we change the destination address
Q.Settings().DestinationAddr((int)(dest_array+0x10));
```

For QDMA, a transfer is initiated when the parameters are loaded into the QDMA registers. This is performed by the Submit() method, which starts the preconfigured transaction, or loads the passed in configuration and submits it.

Only one Qdma transfer may be active in the system at one time. Multi-threaded applications must arbitrate Qdmas as appropriate.

If a terminal count interrupt is not used, a call for WaitForComplete() will delay until the completion occurs. TestComplete() will return a flag that can be used to check completion without blocking.

Qdma transfers may be configured to generate Terminal Count interrupts on completion of the transfer. Which TC bit is signalled is configured in the settings block.

A user supplied handler, similar to an interrupt handler, can be associated with the terminal count interrupt by a call to the TcIntInstall() method. The DMA system shares a single interrupt for all TC interrupts, and the system will call the installed handler when the particular bit in the TC register becomes set. The handler installer requires an Interrupt Binder Object (See “Interrupt Binder Templates” on page 44.) as an argument to associate a handler function or method and argument for the interrupt forwarding mechanism of Pismo. A second function, TcIntDeinstall() removes any installed handler.

Once installed, TC interrupts may be enabled or disabled by a call to TcIntEnable().

The following example shows a full Qdma transfer with TC interrupt handling. In this example a class member function is bound to handle the interrupt response.

```
class DmaIsr
{
public:
    typedef void (*IntFtnType)(void * fallow);

    DmaIsr()
        : Binder(*this, &DmaIsr::MyHandler, NULL)  {  }

    void MyHandler(void * fallow)
    {
        qdma_not_done = false;
    }

    ClassMemberHandler<DmaIsr, void *> Binder;
};

DmaIsr Isr;

void IIMain()
{

    DmaSettings Cfg;
    Cfg.Priority(1).ElementSize(0).SourceIncr(1).DestinationIncr(1);
    Cfg.TCInt(true).TCCode(0);
    Cfg.SourceAddr((int)src_array).DestinationAddr((int)dest_array);
    Cfg.ElementCount(100).ElementIndex(1);
    Cfg.FrameCount(0).FrameIndex(1);
    Qdma Q(Cfg);

    // This QDMA operation will trip a terminal count interrupt when
    // all data has been moved.
    Q.TcIntInstall( Isr.Binder );
    InitArrays();
    Q.TcIntEnable(true);
    qdma_not_done = true;
    Q.Submit();
    while (qdma_not_done)
        ;
}
```

Class Edma. This class manages the posting of EDMA requests. It contains functions to allow configuration of a transfer, initiating a transfer and completion notification via either an interrupt or a polling function. Because the system state is saved in the object, transfers can be predefined and saved to be posted at a later time.

An additional feature of EDMA is the ability to build complicated transfers by linking EDMA transfer blocks or by chaining EDMA transfers together.

For more information on EDMA, see the TI Peripheral Guide.

As with all DMA objects, the Edma object uses one or more internal DmaSettings object to define the transfer. One block is allocated for the primary transfer, and one for each linked block. The Settings() method provdes access to the primary transfer block's settings object. The LinkSettings() similarly allows to one of the link blocks's DmaSettings object. Each of these can be used to call DmaSetting's own configuration functions, or configurations can be loaded from a second object with the Load() method.

```
// Ed is a Edma object, here we change the destination address
Ed.Settings().DestinationAddr((int)(dest_array+0x10));
```

The EDMA transfer can be attached to one of a number of channels. To attach an EDMA to a hardware interrupt, use the channel with the same number as the hardware interrupt. For example, to attach an EDMA to external interrupt 4, use the EDMA channel 4.

For EDMA, before a transfer can be initiated, the parameters are loaded into the EDMA PRAM registers. This is performed by the Submit() method, which loads the PRAM with the transfer information. Unlike QDMA, this does not start the transfer itself. The transfer will be initiated when the associated hardware interrupt occurs. If using software triggering, use the Set() function to initiate a transfer. One Set() call is required for each link block in the transfer.

Each Edma transfer allocates blocks from the PRAM pool to configure its Link blocks. These blocks are a limited resource, and the alloction may fail. If the failure occurs, the IsValid() function will return false.

If a terminal count interrupt is not used, a call for WaitForComplete() will delay until the completion occurs. TestComplete() will return a flag that can be used to check completion without blocking.

Edma transfers may be configured to generate Terminal Count interrupts on completion of any and all blocks in the transfer. Which TC bit is signalled is configured in each settings block. This means there can be different handlers for different blocks in the transfer.

A user supplied handler, similar to an interrupt handler, can be associated with the terminal count interrupt by a call to the TcIntInstall() or LinkTcIntInstall() method. The Link function is used to install a handler for one of the link blocks as opposed to the primary block.

The DMA system shares a single interrupt for all TC interrupts, and the system will call the installed handler when the particular bit in the TC register becomes set. The handler installer requires an Interrupt Binder Object (See “Interrupt Binder Templates” on page 44.) as an argument to associate a handler function or method and argument for the interrupt forwarding mechanism of Pismo. A second pair of functions, TcIntDeinstall() and LinkTcIntDeinstall() removes any installed handler for the TC bit used by the block.

Once installed, TC interrupts for the entire transfer may be enabled or disabled by a call to TcIntEnable().

The following example shows a full Edma transfer with TC interrupt handling. In this example a class member function is bound to handle the interrupt response.

```
class DmaIsr
{
public:
    typedef void (*IntFtnType)(void * fallow);

    DmaIsr()
        : Binder(*this, &DmaIsr::MyHandler, NULL)  {  }

    void MyHandler(void * fallow)
    {
        qdma_not_done = false;
    }

    ClassMemberHandler<DmaIsr, void *> Binder;
};
```

```
DmaIsr Isr;

void EdmaTest()
{
    Edma Ed;

    Ed.Settings().Priority(DmaSettings::priHigh).ElementSize(DmaSet-
tings::is32bit);
    Ed.Settings().ElementIndex(1).ElementCount(50).FrameIndex(1).FrameCount(0);
    Ed.Settings().TCInt(true).TCCode(1).FrameSync(true);
    Ed.Settings().SourceAddr(int(&src_array[0])).SourceIncr(DmaSettings::Incr);
    Ed.Settings().DestinationAddr(dest_array).DestinationIncr(DmaSettings::Incr);
    //
    //  Define a linked
    DmaSettings Cfg;

    Cfg.Priority(1).ElementSize(0).SourceIncr(1).DestinationIncr(1);
    Cfg.TCInt(true).TCCode(1).FrameSync(true);
    Cfg.SourceAddr((int)&src_array[0]).DestinationAddr((int)(dest_array+50));
    Cfg.ElementCount(50).ElementIndex(1);
    Cfg.FrameCount(0).FrameIndex(1);

    Ed.AddLink(Cfg);
    Ed.LinkTcIntInstall( 0, Isr.Binder );
    Ed.TcIntClear();

    // This EDMA operation will trip a terminal count interrupt when
    // all data has been moved.
    InitArrays();
    Ed.TcIntEnable(true);
    qdma_not_done = true;
    Ed.Submit();
    // We software-initiate the EDMA here, but if this EDMA were using EINT4..7,
    // then an external int hardware pulse would remove need for Ed.Set, below
    Ed.Set();
    while (qdma_not_done)
        ;
    // Need to sync L2 cache with the of SDRAM, so that CPU can see the data
    CACHE_clean(CACHE_L2, dest_array, sizeof(dest_array));
    //
    //  Transfer the second transfer block...
    Ed.Set();
    while (qdma_not_done)
        ;
    // Need to sync L2 cache with the of SDRAM, so that CPU can see the data
    CACHE_clean(CACHE_L2, dest_array, sizeof(dest_array));
}
```

The above example sets up a two block linked transfer triggered by software. A TC Interrupt is configured to signal the completion of each block in the transfer. The mainline waits for each block transfer to finish, as notified by the interrupt handler. Then the next block transfer is triggered by a second call to Set(). The Cache functions are required to assure that the cache and memory contents are back in synchronization.

Linked and Chained blocks. EDMA transfers may span multiple transfer blocks. On the completion of the primary transfer, the first link block is loaded into the primary block and initiated. When this block completes, the next linked block is loaded, and so on. A link block can form a loop, but it is important to remember that the primary block can never be part of a loop. Since it is overwritten by the first linked transfer, this transfer can only occur once. Because of this to make a loop of two transfers

requires **three** blocks to be configured. The primary block contains the first transfer, the first link the second transfer, and the third is a repeat of the first transfer that is linked back to the first link block.

Link blocks are allocated by a call to AddLink(). This call automatically configures the preceding block to link to this newly added block. It returns the index of the newly added block that can be used in order to configure the link block. To form a closed loop in a block chain, call LinkBackTo(). This connects the final block in the chain back to the block whose index is given in the argument.

Transfer chaining is a mechanism for having a transfer trigger another on completion. The ChainTo() and ChainEnable() methods set up a chaining relation between two transfers. Note that on the TI C671x processor, the second transfer must be configured on channels 8-11.

Class EdmaMaster. This class acts as a holder for functions and information common to all EDMA interrupts instead of associated with a single EDMA channel. Only one instance of EdmaMaster is created at program initialization. It is accessed by calling the static member function EdmaMaster::Object().

EdmaMaster contains several functions dealing with the EDMA PRAM. This is a memory region shared among all EDMA objects giving a common storage for configuration blocks. This is a limited resource, so be wary of allocating many Edma blocks and not releasing them. The method ClearPram() clears all the PRAM blocks in a single operation.

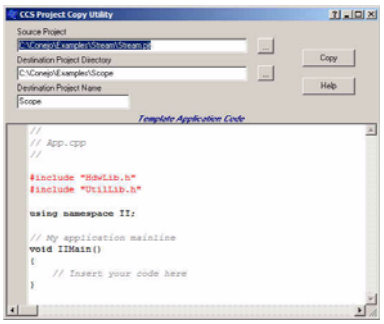
EdmaMaster contains several functions dealing with the EDMA PRAM. This is a memory region shared among all EDMA objects giving a common storage for configuration blocks. This is a limited resource, so be wary of allocating many Edma blocks and not releasing them. Also available are functions to give access to the area at the end of the PRAM that is not used by the system. This scratchpad memory might be of use as a shared memory pool in an application.

CHAPTER 4

Building a Target DSP Project

Building a Target DSP Project

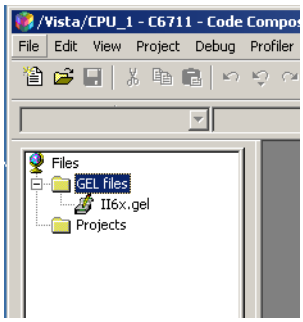
Building a project suitable for an M6713 baseboard requires a particular setup of the project.



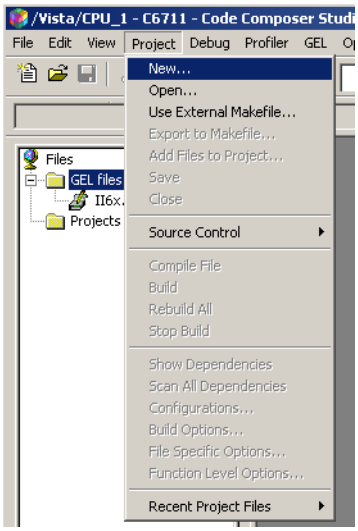
By far, the easiest way to create a new DSP project is by using an existing project as a template. The `CopyCcsProject` applet provided in the Pismo Toolset automates this task. To use this utility, select an existing Code Composer project as the Source Project, typically one of the example programs supplied in the Pismo Toolset. Next, select the directory into which you wish the new project to be created, using the Destination Project Directory edit control. Then, edit the Destination Project Name for the newly-created project. Finally, click the Copy button to create the new project from the template. The new project may be opened and used within Code Composer.

Alternately, you may follow the manual steps below to create a new target DSP project. The project name used below is called `Test`, but you should name your project appropriately for your application.

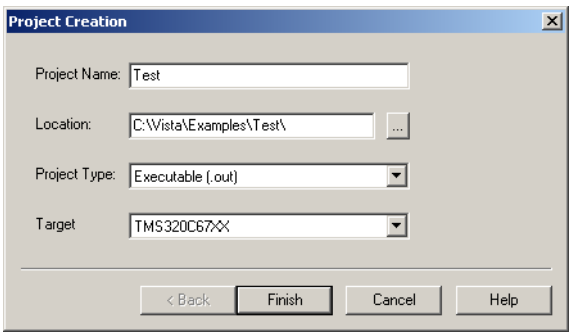
- Start Code Composer Studio. In the default configuration, the project window will contain no projects but will contain the default Innovative-supplied board initialization GEL file.



- Click Project | New on the menu bar to create a new DSP project.

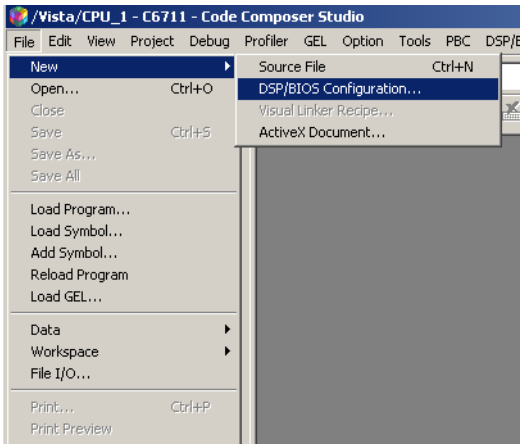


- Specify the location for the new project and its name. In this example, a new project called `Test` is being created in the `M6713\\Examples\\` directory. Change the location to accomodate your board type and processor type.

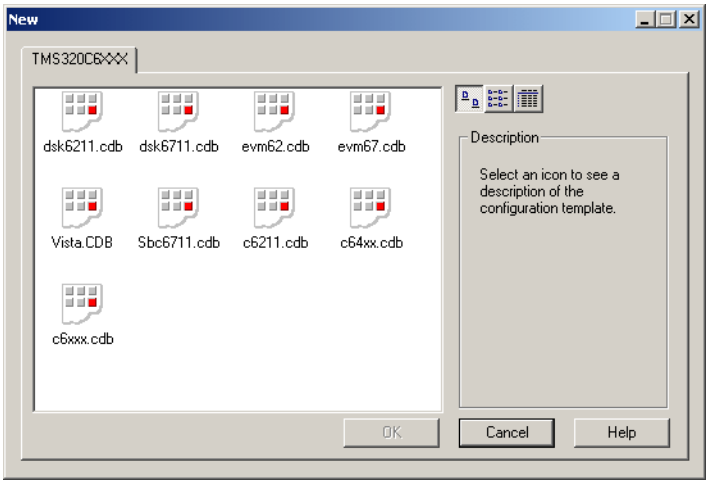


- After the new project has been created, it will appear in the CCS project window under the Projects folder.

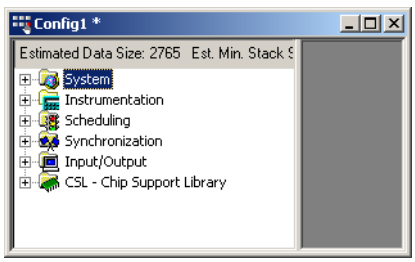
- Click `File | New | DSP/BIOS Configuration` to create a new CDB file for use in the project.



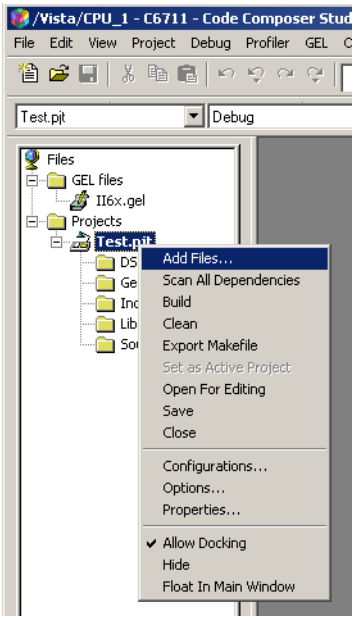
- Select the template for the baseboard from the list of CDBs in the New CDB dialog box. This CDB is named after the baseboard type. For example, for the M6713 choose `M6713.CDB`, and for the Conejo baseboard choose the `Conejo.CDB` template



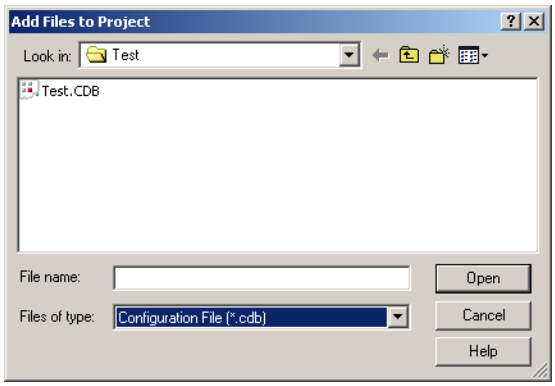
- By default, this CDB will be named Config1. Save it as `Test.CDB`.



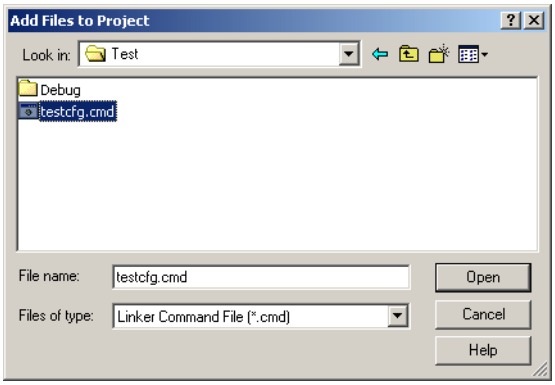
- Though the CDB and its support files have been created on disk, you must manually add them to the Test project. Right-click on `Test.pjt` in the Project window to invoke the project hot menu. Click `Add Files` to add a file to the project.



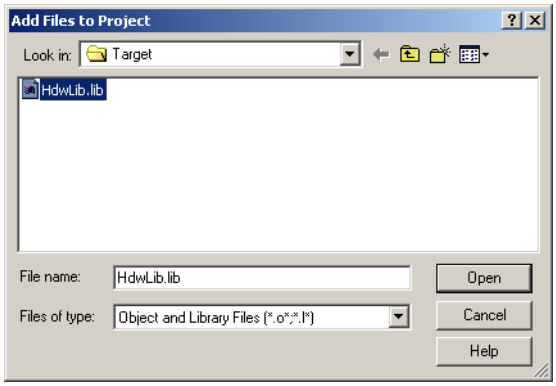
- Select the the newly-created `Test.cdb` for addition to the project. This will implicitly add the auto-generated files `Testcfg.s62` (`Testcfg.s64` for Velocia cards) and `Testcfg_c.c` to the project as well.



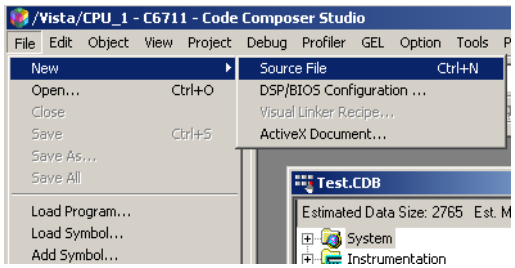
- Right-click on `Test.pjt` in the project window, click “Add Files” then select the the newly-created `Test.cmd` for addition to the project.



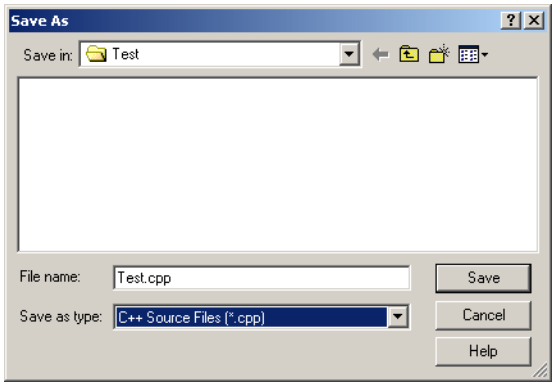
- Right-click on `Test.pjt` in the project window, select `Add Files`, then browse to the Examples directory and select `Examples.cmd` for addition to the project.



- Add an new C++ source file to the project. `Click File | New | Source File` to create an empty source document.



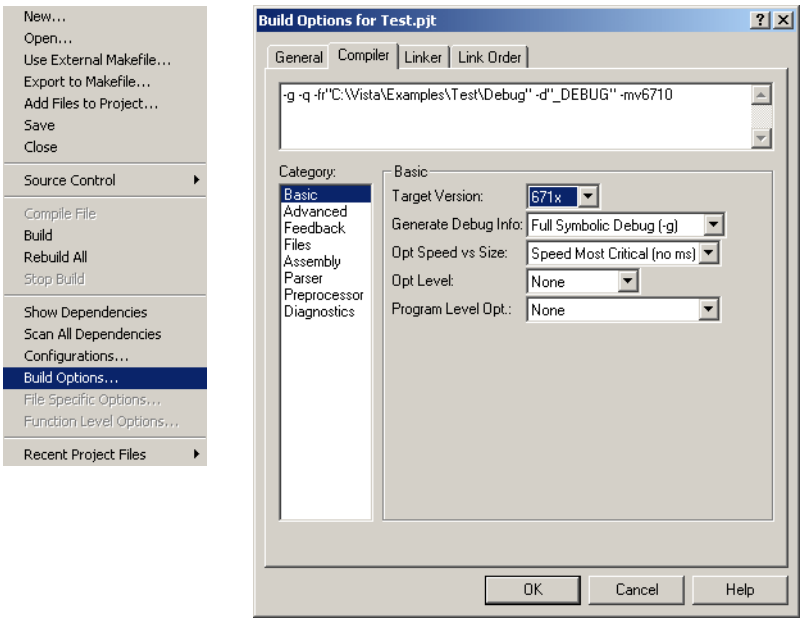
- Rename the new source document to `Test.cpp`. To use the Pismo libraries, you must use C++ files and the C++ compiler, even if you intend to restrict your own coding to the C subset of C++



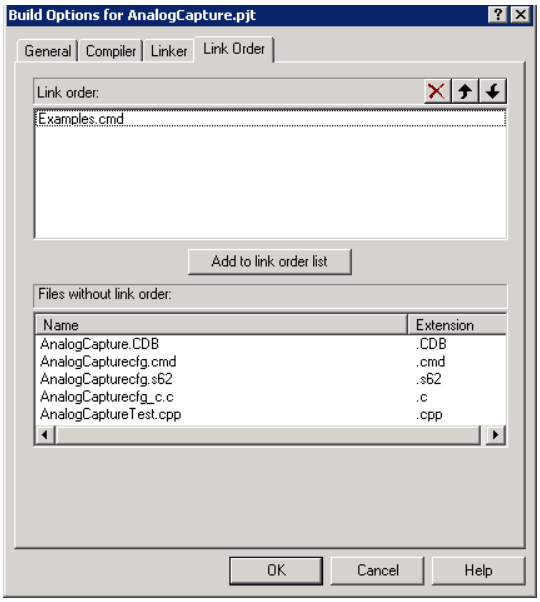
- Type the boilerplate code below into your source file. This is the minimum code needed for any Pismo C++ application.

```
//
// App.cpp
//

#include "Pismo.h"

using namespace II;

// Application mainline
void IIMain()
{
    // Insert your code here
}
```

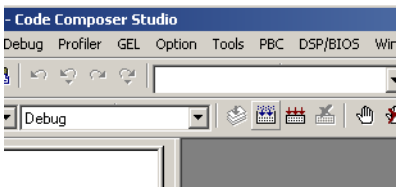
- Click the menu `Project | Build Options` to invoke the compiler `Build Options` dialog. Then select the **Files** Category, then enter the pathspec to the `Examples.opt` file in the Examples directory to the Options File edit box.



- Click on the Link Order tab, then add `Examples.cmd` to the Link Order List.



- Click the Incremental Build button to rebuild the template application. It should compile and link without errors.



Writing a Program

The basic program given in the example above includes a 'Main' function *IIMain()*. DSP/BIOS, the OS used in the Pismo library, uses code inserted after exiting from the normal C-language *main()* to initialize features of DSP/BIOS. This means that some language features are not available then. To avoid these problems the Pismo library provides a *main()* function and uses it to create a single thread. This thread, when executed, calls the *IIMain()* function. Inside of this thread, all DSP/BIOS is initialized and ready for use. It is required that the user include this function, and use it as the equivalent of the old main process function in C.

Host Tools for Target Application Development

The Innovative Integration Pismo Toolset allows users of Innovative DSP processor boards to develop complete executable applications suitable for use on the target platform. The environment suite consists of the TI Optimizing C++ Compiler, Assembler and Linker, the Code Composer debugger and code authoring environment as well as Innovative's custom Windows applets (such as the terminal emulator).

Code Composer Studio is the package used to automate executable build operations within Innovative's Pismo Toolsets, simplifying the edit-compile-test cycle. Source is edited, compiled, and built within

Code Composer Studio, then downloaded to the target and tested within either the Code Composer Studio debugger or via the terminal emulator.

Code Composer Studio may be used for both code authoring and code debugging. Details of constructing projects for use on Innovative DSP platforms are given in the above section of this chapter.

Do not confuse the creation of target applications (code running on the target DSP processor) with the creation of host applications (code running on the host platform). The TI tools generate code for the TI DSP processors, and are a separate toolset from that needed to create applications for the host platform (which would consist of some native compiler for the host processor, such as Microsoft's Visual C++ or Borland Builder C++ for IBM compatibles). To create a completely turnkey application with custom target and host software, *two* programs must be written for *two* separate compilers. While Innovative supports the use of Microsoft C/C++ for generation of host applications under Windows with sample applications and libraries, we do not supply the host tools as part of the Development Environment. For more information on creating host applications, see the section in this manual on host code development.

This section supplies information on the use of the development environment in creating custom or semicustom target DSP software. It is not intended as a primer on the C++ language. For information on C/C++ language basics, consult one of the primer books available at your local bookstore.

Components of Target Code (.cpp, .cdb, .cmd, .pjt)

In general, DSP applications written in TI C++ require at least three files: a .cpp file (or "source" file) containing the C++ source code for the application a .cmd file (or "command" file) which contains the target-specific memory-map and build data needed by the linker, a .cdb file (or "command database" file) which specifies the properties of the BIOS operating system used within the application and a .pjt file ("project" file) which centralizes all project-specific options, settings and files. There may also be one or more .asm assembler source files, if the user has coded any portions of the application in assembly language.

Edit-Compile-Test Cycle using Code Composer Studio

Nearly every computer programming effort can be broken down into a three step cycle commonly known as the edit-compile-test cycle. Each iteration of the cycle involves editing the source (either to create the original code or modify existing code), followed by compiling (which compiles the source and creates, or builds, the executable object file), and finally downloading and testing the result to see if it functions in the desired fashion. In the Innovative Intergration development system these stages are accomplished within the Code Composer integrated development environment (IDE).

By using Code Composer Studio, these stages of the programming cycle are accomplished entirely within the IDE. The project features of Code Composer Studio support component file editing and compilation stages, along with allowing the executable result to be downloaded and tested on the target hardware. This fully integrated programmers environment is more user-friendly then the basic command line interface, which comes standard with the TI tools.

Automatic projectfile creation

When a project is created, opened, modified, built or rebuilt, the Code Composer Studio dependency generator automatically generates a project makefile (named `<project file>.pjt`, located in the project directory), which is capable of rebuilding the project's output file from its components.

This file is automatically submitted to the internal make facility whenever you click on build or rebuild within Code Composer Studio. The make facility automatically constructs the output file by recompiling the out-of-date source files including the dependencies contained within those source files.

Rebuilding a Project

It is sometimes necessary to force a complete rebuild of an output file manually, such as when you change optimization levels within a project. To force a project rebuild, select `Project | Rebuild All` from the Code Composer Studio menu bar.

IIMain replaces main. Due to restrictions within Dsp/Bios, not all BIOS features may be safely used within main(), since it is called early in the system initialization sequence. To circumvent this limitation, Pismo automatically constructs a default thread running within normal priority and starts this thread automatically. The entry point function in this thread is called `IIMain`, and all Pismo applications must define this function. This function is intended to replace main in your application programs. You may safely call any BIOS function within `IIMain`.

Running the Target Executable

The `test` program may be converted into a simple, "Hello World!" example, by using the built-in standard I/O features within Pismo. Bring up the `Test.cpp` source file edit screen. Scroll down the source file by using cursor down button until you reach the `IIMain()` function. Edit it as follows:

```
#include "Pismo.h"

cio << init;
cio << "Hello World!" << endl;
cio.monitor();
```

You can now compile the new version by executing Build from the Project menu (or by clicking on its toolbar icon). This causes Code Composer Studio to start the compiler, which produces an assembly language output. The compiler then automatically starts the assembler, which produces a `.obj` output file (`test.obj`). Code Composer Studio then invokes the TI Linker using the `testcfg.cmd` file, which is located in the project directory. This rebuilds the executable file using the newly revised `test.obj` . If no errors were encountered, this process creates the downloadable COFF file `test.out`, which can be run on the target board. At this point, the program may be run using the terminal emulator applet, which may be invoked using the terminal emulator shortcut located within the target board program group created during the Pismo Libraries installation process. In the terminal emulator, download the `test.out` file. The program runs and outputs the message "Hello, World" to the terminal emulator window.

If errors are encountered in the process, Code Composer Studio detects them and places them in the build output window. If the error occurred in the compiler or assembler (such as a C++ syntax error), the cursor may be moved to the offending line by simply double-clicking on the error line within the build output window, and the error message will be displayed in the Code Composer Studio status bar. If the linker returns a build error, the build output window shows the error file. From this information, the linker failure can be determined and corrected. For example, if a function name in a call is mis-

spelled, the linker will fail to resolve the reference during link time and will error out. This error will be displayed on the screen in the build output window.

Note: Be sure to start the terminal emulator **BEFORE** starting Code Composer, to avoid resetting the DSP target in the midst of the debugging session. If the terminal emulator is not yet running and you wish to run the Test object file, perform the following steps.

1. Execute Debug | Run Free to logically disconnect the DSP from the debugger software.
2. Terminate the Code Composer Studio application.
3. Invoke the terminal emulator application.
4. Restart the Code Composer Studio application.

This outlines the basics of how to recompile the existing sample programs within the Code Composer Studio environment.

Anatomy of a Target Program

While not providing much in the way of functionality, the `test` program does demonstrate the code sequence necessary to properly initialization the target. The exact coding, however, is very specific to the I.I. C Development Environment, target boards, and is explained in this section in order to acquaint developers with the basic syntax of a typical application program.

```
/*
*       HELLO.CPP
*       Test file/program for target board.
*/

#include "hdwlib.h"
#include "utillib.h"

IIMain()
{
        cio << init;

        cio << “Hello World!” << endl;
        cio << “\nEchoing keystrokes...” << endl;

        char key;

        do
                {
                cio >> key;
                cio << key << flush;
                }
        while(key != 0x1b);
```

```
        cio.monitor();
}
```

The two lines of the program that being with a "#" are #include statements, which include the header files for the hardware and utility I/O libraries. These include prototypes for all the library classes within Pismo.

The `cio << init` invokation will setup the standard monitor I/O interface and reset the terminal window. The next lines perform the basic standard I/O functions of printing "Hello World!" & "Echoing keystrokes...". These two lines are where custom code could be inserted.

The following do-loop sequence simply echoes keys typed at the terminal emulator back to the terminal display, until the Esc key is pressed. When Esc is pressed, the `cio.monitor()` function effectively terminates the program, except that interrupts are still active and interrupt handlers (if they had been installed) would still execute properly.

The `test` program is very simple, but it contains the basic components of a typical DSP application, as well as the initialization needed to interact with the terminal emulator.

Use of Library Code

Library routines can be compiled and linked into your custom software simply by making the appropriate call in the source and adding the appropriate library to the linker command file. Refer to the library reference within the Pismo online help for library location information on each class and method.

In general, user software needs to `#include` the relevant library header file in source code. The header files define prototypes for all library functions as well as definitions for various data structures used by the library functions. The files `HdwLib.h` and `UtilLib.h` should be included within all programs; The file `DspLib.h` should be included if a program uses functions in the DspLib signal processing library.

Example Programs

Under **<baseboard>\Examples** in the install directory, the baseboard's example programs are installed. Some examples have no host component, and some use the terminal emulator applet as the host. Host examples are written in C++ either under Borland Builder or Microsoft MSVC, or both. Target examples are written using CCS 2.x and DSP/BIOS. Note that not all of the examples listed below are available for all targets.

TABLE 10. Pismo Example Programs

Example	Host	Target	Illustrates
AEcho	terminal emulator	DSP/BIOS	Use of DSP/BIOS drivers; Analog output driven from analog input
AnalogIn	terminal emulator	DSP/BIOS	Analog capture into DSP memory. Rate limited by Omnibus interface.

TABLE 10. Pismo Example Programs

Example	Host	Target	Illustrates
A16D2AnalogIn	terminal emulator	DSP/BIOS	Analog capture into DSP memory. Tailored version of AnalogIn example to exploit special mux features of A16D2 module.
AnalogOut	terminal emulator	DSP/BIOS	Analog waveform playback from buffer in DSP memory. Rate limited by Omnibus interface.
AnalogCapture	terminal emulator	DSP/BIOS	Full-rate analog capture to FIFO memory on suitably-equipped Omnibus modules
AWave	terminal emulator	DSP/BIOS	Analog output driven by local signal generator objects.
FftFloat	terminal emulator	DSP/BIOS	Use of Fourier class to perform foward and inverse FFTs
FirFloat	terminal emulator	DSP/BIOS	Use of BlockFir class to perform FIR filter functions.
Edma	terminal emulator	DSP/BIOS	Use of Pismo Edma and Qdma wrapper classes with installable interrupt handlers.
Files	terminal emulator	DSP/BIOS	Use of C++ Standard I/O library
DataXfer	BCB MSVC	DSP/BIOS	Use of Pismo Host<->Target message and data packet passing via PCI bus interface.
FpdpEcho	terminal emulator	DSP/BIOS	Use of FPDP driver to flow data through loopback connector.
FpdpPio			Bit control of FPDP port via driver methods
DisplayTest	terminal emulator	DSP/BIOS	Features of RamTex graphics library for OLED display in TOPO enclosure
Numeric\Test	terminal emulator	DSP/BIOS	Features of numeric math library - curve fitting, statistics, etc
Servo	terminal emulator	DSP/BIOS	Servo application example using Servo class. (Not available for Vista or Delfin)
Swi	terminal emulator	DSP/BIOS	Use of Pismo SoftInt class for software interrupts.
Timer	terminal emulator	DSP/BIOS	Use of Pismo ClockBase objects for timebase control.
DioData	terminal emulator	DSP/BIOS	Use of baseboard digital I/O

The Next Step: Developing Custom Code

In building custom code for an application, Innovative Intergration recommends that you begin with one of the sample programs as an example and extend it to serve the exact needs of the particular job. Since each of the example programs illustrates a basic data acquisition or DSP task integrated into the target hardware, it should be fairly straightforward to find an example which roughly approximates the basic operation of the application. It is recommended that you familiarize yourself with the sample programs provided. The sample programs will provide a skeleton for the fully custom application, and ease a lot of the target integration work by providing hooks into the peripheral libraries and devices themselves.

CHAPTER 5 *Servo Applications*

Applications for DSP baseboards can be broadly divided into two categories: Data Acquisition and Servoing applications. These two types of application require very different methods of behavior by the system that cannot easily be reconciled into a single 'one size fits all' system. Yet the Pismo library needs to support both styles of program in a simple and natural way.

Data Acquisition Applications. Acquisition applications need to acquire large quantities of data, but do not need to process the data immediately. In order to support higher rates, these applications use large buffers, and large chains of these buffers to allow data to be moved from the hardware into memory without CPU intervention. Thus data can be acquired at high rates, but the system only needs to be involved at relatively rare intervals to process a block of data.

These applications are very natural for DMA, and often DMA is used to move this data into memory. In addition, these buffers and the FIFOs in the hardware allow some 'slack' in the system so that the application can fall behind for a short time if it is busy performing some other service. After the completion of this task the system can process the buffers in the queues and in the hardware FIFOs and catch up. As long as enough slack is built into the system, no data loss will result.

Servoing Applications. In a Servo application, the requirements are polar opposites from the Data Acquisition application. In this case, each event needs to be processed at once, with no delay. This data is analyzed to produce an output update event that is output to the DACs in the system at once. It is therefore vital that data is read into the system and processed without any buffering. Hardware FIFOs are only useful at the single event level -- if you fall behind by even a single event your servo is failing.

Since the amount of data moved from the hardware is so small, DMA is much less useful in this case than it is for the Data Acquisition case. In fact, due to very large latencies in initiating a DMA operation, only quick DMA (QDMA) is a reasonable

candidate for use within a servo application. However, QDMA is not utilized within the Pismo servo driver for the following reasons:

1. QDMA requires software initialization in order to initate data movement. This initialization must be performed each servo interrupt cycle. The amount of time saved by having the DMA engine (more efficiently) move between the peripheral and the servo application is roughly comparable to the amount of time lost performing QDMA setup and cache manipulation functions.
2. QDMA is a limited system resource. The Pismo Buffer classes use QDMA to perform high speed copies. So, use of QDMA within the servo driver would preclude its use elsewhere.

Servoing and DSP/BIOS. DSP/BIOS uses device drivers to provide a uniform, simple interface to analog hardware for input and output. The driver model used is well suited for Data Acquisition applications, since it provides simple means for allocating and passing buffers of data in and out of the device. This driver model is used for all of the streamable hardware provided on the baseboard. Yet because of the strength of the model for the streaming applications, these drivers fail to be able to properly servo in any efficient manner.

The Servo Base Class. The Pismo Library supports servoing by defining a class to perform the basics of a servo operation. The basic servo operation is to acquire data from the analog input, perform some kind of algorithm using the new data to produce an output data event, which is then output to the DACs. The basic servo class handles the details of attaching the interrupt, setting up the hardware, reading the data from the hardware and delivering it to the user's function for processing. After this, it writes the modified data to the DACs.

Using the Servo Class

The `ServoBase` base class provides a number of methods to simplify the operation of a servoing application. It organizes the configuration of the hardware and the timebases to provide the exact setup needed. It provides a 'callback' hook to allow user code to be called when running the servo in order to provide custom servo processing functionality.

The following table gives the methods that can be overridden, and when they are called:

TABLE 11. The Servo Class Virtual Function

Function	When Called	Function
`Execute()`	On each interrupt	Servo calculations

Servo Interrupt Modes

The `ServoBase` class sets up a configuration where the analog will drive an interrupt on every incoming point, and expect a return event to be loaded very soon thereafter. A key factor in the performance of the servo is the performance of the interrupt and its internal functions.

The *`ServoBase`* class provides all the functionality of a 'null' servo...that is, one that performs no modification of the data from input to output, including installation of its own interrupt handler. This handler will read data from the input peripherals into a buffer, call the `Execute` function in which the user performs custum servo calculations, then transfers the data out to the output peripherals.

ExecuteUser code within the overridden `Execute` method always executes within HWI interrupt context. This code may call BIOS routines only if the the ServoBase object used within the application is constructed with the `UseDispatcher` parameter set `true`. However, setting this flag results in a less-efficient interrupt handler, exhibiting high interrupt latencies than if `UseDispatcher` is `false`.

Once in the interrupt, the application software merely manipulates samples within the event buffer whose address is passed as a parameter to the Execute method. Input samples are consumed from this event buffer and output response values are stored into this same buffer. Upon return from the overridden Execute method, the contents of this buffer is automatically written to the output devices.

The use of these methods allows the servo system overhead to be reduced to the minimum possible for this architecture. The Servo example program, included in the Developers Package, demonstrates this technique.

Execute()

The `Execute` method is overridden with the user's Servo code. The function is very simple -- the data taken from all enabled input peripherals is passed in as an argument, and needs to be overwritten with the data to be written to all enabled output peripherals. The data is passed in the same format as the peripherals produce -- for example, paired data in channel order on the Servo16 module. Similary, the output format is that which the output peripherals will require. If the channels output are different than those input, or more or fewer, the data is stored left-justified within the event buffer for proper results to be obtained.

This may seem complicated, but it is really rather simple in practice. The job of this function is to take the input data and 'convert' it into an output event using your own algorithm. All other details are taken care of by the system.

Timebases

The Servo object performs analog input and analog output, just as device drivers do, so it needs to allow timebase information to be passed into it. A clock user interface object is manipulated in a fashion identical to the streaming analog drivers to initialize the servo conversion clock.

DAC Delay

Some modules, such as the Servo16, support a mode which delays the conversion of the DACs a fixed amount relative to the conversion clock. The purpose of this delay is to give time for the Servo interrupt processing code to acquire, analyze, and produce the output data for the DAC to output. This value needs to be tuned to match the time it takes your specific application to do this operation, depending on the servo code. The delay is set by the Delay() property of the Servo object.

Channel Selection

The ServoBase objects InputChannels() and OutputChannels() methods are used to obtain access to the channel information objects for the module's input and output peripherals. Generally, the use of these methods is to specify the number of active input and output channels.

Servo Timing

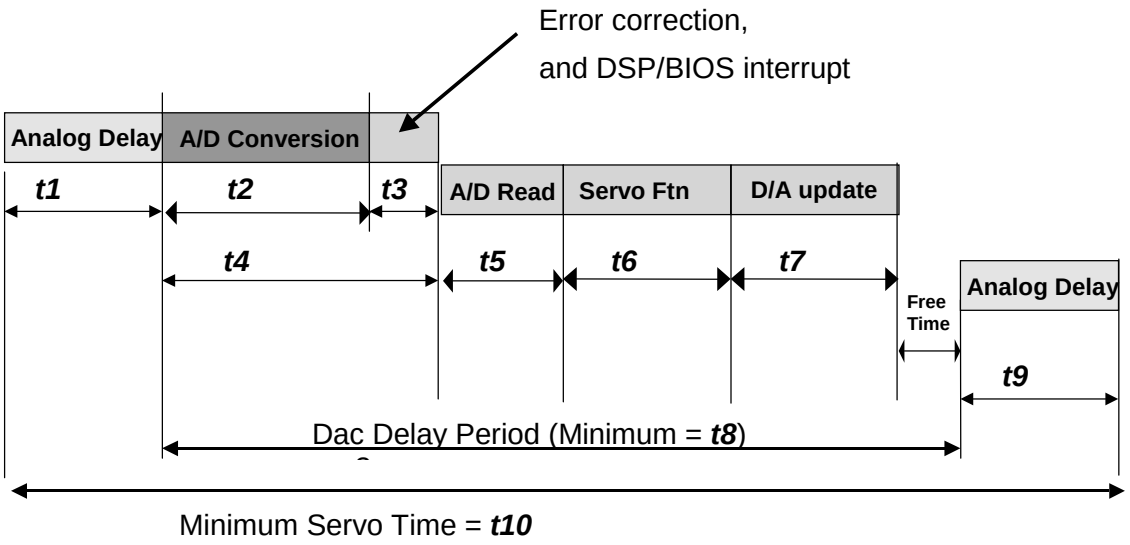


TABLE 12. Servo Time Measurements

		Servo16 Module	
Interval	**Purpose**	**Period in μS (w/Dispatcher)**	**Period in μS (w/o Dispatcher)**
t1	Adc Analog Delay	15.0	
t2	Adc Conversion	10.0	
t3	Logic + Interrupt Time	0.8 + 2.9	0.8 + 1.0
t4	***t2*** + ***t3***	13.7	11.8
t5	A/D Read Time	3.1 for 8 pairs 0.8 for 1 pair	
t6	Servo Calculation Time	?	
t7	D/A Write and Update plus module interrupt acknowl-edge	4.8 for 5..8 pairs 3.2 for 3..4 pairs 2.3 for 1..2 pairs	
t8	Minimum Dac Delay Time	16.8..21.6	14.9..19.7
t9	Dac Analog Delay	15.0	
t10	Total Servo Time	31.8..36.6	29.9..34.7.

The analog delay inserts a fixed skew between input and output and does not affect the update rate directly.

A Servo Tutorial

The following section walks through an example servo. This example can be found in the distribution in the Servo directory under the Examples directory.

This servo modifies one channel in a simple but visible way to show the servo effect. In actual practice all channels would probably be modified.

The first interesting operation is the derivation of a new ServoBase class.

This example does not utilize the non-dispatcher, to maximize performance. The TestServo class informs the base class that the application will not make BIOS calls within the interrupt by the second 'false' argument to the constructor.

```
//============================================================================
//  class TestServo -- this examples servo
//============================================================================

class TestServo : public ServoBase
{
public:
    TestServo(Omnibus::ModuleSite site)
        : Servo(site, false)
        {}

};

};
```

To derive, you need to create a constructor for the new class. Since the ServoBase base class requires an argument, C++ can not create one for you. The base class needs to be informed of which Omnibus module site is hosting the installed Servo16 I/O module. In this case, we require our servo to have it passed in when it is created. An alternative would be to 'hard code' a value and not require (or allow) the user to change it.

Next, we define our interrupt handler function.

```
// Overrides
#pragma CODE_SECTION(".hwi");
virtual void Execute(volatile short * event, int inputs, int outputs)
{
    // clip above this amplitude
    const int Threshold = 0x4000;

    if (event[0] > Threshold)
        event[0] = Threshold;
}
```

Here we actually define the servo function. The handler is defined as an override for the virtual `Execute` method. The `#pragma` places this function onchip for extra efficiency.

The base class passes several important parameters to the application. `event` is a pointer to a buffer containing an array of 16-bit input samples read from the input peripheral obtained during the most recent conversion. `inputs` and `outputs` are the count of active input and output channels, respectively. The body of the method illustrates processing on input channel zero only. The value of input channel zero is clipped to 0x4000 counts. For the Servo16, this clips the input voltage to half-scale. The clipped value is written back to the event buffer, overwritting the original value. The effect of this handler is to loop-through all inputs unchanged to all outputs, except for channel zero, which is clipped.

The Pismo Library creates a new `main()` function, `IIMain()` which is the primary application thread. This thread is responsible for creating, configuring, and running the servo.

```
//---------------------------------------------------------------------------
//  IIMain()  --  Application mainline
//---------------------------------------------------------------------------

void IIMain()
{
    int DacDelay;
    int SampleRate;

    //
    //  Terminal I/O
    cio << init;
    cio.At(Point(25, 0));
    cio << bold << "\7Servo Application\n\n" << normal << endl;

    TestServo ServoIo(Omnibus::mSite0);
```

First, the application instantiates a `TestServo` object called `ServoIo`. This provides access to the servo driver, customized with a unique `Execute` behaviour. This driver is `Open`ed, then the internal event buffer is relocated to onchip ram, for enhanced performance.

```
    LoadModule(Omnibus::mSite0, PickModuleType());

    bool status = ServoIo.Open();
    cio << "Servo open " << (status ? "ok" : "failed") << endl;
    if (!status)
        DynamicHang("Servo interface not supported");

    // Locate event buffer onchip
    ServoIo.SegId(1);
```

First, the application obtains a pointer to the timebase used for the analog input. This is a ‘plain old timebase’ which runs continuously with no special triggering. Its rate is set by the property call to `Clock->Rate()`.

```
    cio << "Enter sample rate (<= 100 kHz): " << flush;
    cio >> SampleRate;
    cio << endl;
    SampleRate *= 1000;

    ClockRateUI * Clock = ClockRateUIPtr(ServoIo.Clock());
    if (Clock)
        Clock->Rate(SampleRate);
```

The example allows a variable number of channels be processed. This is accomplished by manipulating the input and output channel control objects accessible through the `InputChannels` and `OutputChannels` methods.

```
    int MaxChannels = std::min(ServoIo.InputChannels().Channels(),
                  ServoIo.OutputChannels().Channels());
    int Channels;
    cio << "Enter # channels: " << flush;
    cio >> Channels;
    cio << endl;
    Channels = std::min(MaxChannels, Channels);

    // Enable all analog input and output channels
    ServoIo.InputChannels().EnableChannels(Channels);
```

```
ServoIo.OutputChannels().EnableChannels(Channels);
```

The output delay is also adjustable, through use of the `Delay` method inherited from the base ServoBase class.

```
int MaxDelay = (((1.0/SampleRate) * 1.e9) / 100) * 100;
cio << "Enter the Dac Delay Time (< " << MaxDelay << " ns): " << flush;
cio >> DacDelay;
cio << endl;
//
//  ...Configure delay of DAC clock
ServoIo.Delay(DacDelay);
```

Interrupt latency is increasingly erratic in the presence of other interrupt sources. The example allows the USB and serial port interrupt to be disabled, to improve determinicity.

```
char NoComm;
cio << "Disable communications? " << flush;
cio >> NoComm;
cio << endl;

cio << "\nServo Running... press any key to stop." << endl;
Sleep(300);

// If communications are disabled, max servo interrupt latency is
// reduced, since fewer interrupt conflicts occur
if (std::tolower(NoComm) == 'y')
    DisableCommunications();
```

The driver is open and viable. To start data flow, simply call the `Start` method. Interrupt processing within the `ServoIo Execute` method will commence immediately thereafter.

```
// Start processing
ServoIo.Start();
while (!cio.KbdHit())
    {
    Sleep(300);
    cio << "Serviced interrupt: " << ServoIo.Tally() << cr;
    }
```

Data flow may be suspended and resumed through repeated calls to `Start` and `Stop`.

```
// Stop processing
ServoIo.Stop();
// Close driver
status = ServoIo.Close();

cio << "\nProgram terminating" << endl;
cio.monitor();
```

The loop continues to run until the user hits a key on the operating panel. The main thread must be kept from terminating to keep the Servo alive. The above code is a simple way to do it.

Then when the loop is finished, we close down and exit:

```
// Terminate streaming
status = ServoIo.Close();
```

Debugging Note: It is important that the `Close()` method be called during debugging, or that the CPU be reset for proper servo results. We have found that if the servo is interrupted and the program reloaded

in CodeComposer directly, the servo may be delayed an additional clock cycle. Either performing a full download from the host program or allowing the servo to close avoids this problem.

Conclusion. The power of the Pismo library and the servo application is that the details of configuring the analog are hidden in the objects provided, yet can be configured if necessary. The code that is written is much clearer since the support code is concealed internally. The application is simpler and easier to maintain because the code is almost entirely the important servo algorithm.

CHAPTER 6

Communication to the Host

Overview

Many applications involve communication with the host CPU in some manner. All applications at a minimum must be reset and downloaded from the host, even if they run independently from the host after that.

Other applications need to interact with a host program during the lifetime of the program. This may vary from a small amount of information to acquiring large amounts of data. Some examples:

- Passing parameters to the program at start time
- Receiving progress information and results from the application.
- Passing updated parameters during the run of the program, such as the frequency and amplitude of a wave to be produced on the target.
- Receiving alert information from the target.
- Receiving snapshots of data from the target.
- Sending a sample waveform to be generated to the target.
- Receiving full rate data.
- Sending data to be streamed at full rate.

These different requirements require different levels of support to efficiently accomplish. The simplest method supported is performing file I/O from within Code Composer using either the standard C file functions (which communicate directly through CCS to the Host file system), or via the Innovative terminal emula-

tor, which supports simple data input and control and the sending of text strings to the user in addition to file I/O.

However, the highest level of support is given by the Bus Mastering Interface. This allows sophisticated high-rate transfer of commands and information between the host and target. It requires more software support on the host than the standard I/O does.

CPU Busmastering Interface

Firmware within the logic onboard the M6713 is capable of high-speed PCI busmastering to move data between target and host memory. This busmaster facility can be used to transfer data between host and target applications. Enhanced DMA is used to move data between DSP memory and a logic-based FIFO. The firmware within the baseboard logic moves data between this FIFO and Host PC memory.

CPU Busmastering Implementation

Packet Based Transfers. The CPU busmaster interface implemented within the the M6713 transfers discrete blocks between the source and destination. Each data buffer is transferred completely to the destination in a single operation. Only if several transfers are requested at once will any delay in beginning transmission occur, as multiple requests have to be serialized through a single hardware system.

The data buffers transferred can be of different sizes. Each requested buffer is interrogated for its size and fully transmitted. At the destination, the destination buffer is dynamically re-sized to allow the incoming data to fit. If the buffer given is too small for the data, it will be reallocated to allow the transfer. Reallocating buffers can take some time, for best performance buffers should be pre-sized to be large enough for the largest transfer expected. This will make allocation of buffers at critical times unnecessary.

Blocking Interface. CPU busmastering uses a simple blocking interface for its send and receiving functions. The sending function will not return until the transfer has completed and the buffer is ready for reuse. Similarly, the receiving function waits until data has arrived from the data source and transferred into the data buffer before returning. At this point the buffer is ready for use. This blocking allows sequences of transfers managed by a simple sequence of calls to transfer functions.

Since the transfer functions are blocking, they are best avoided in the main user interface thread of a Windows application. The GUI will be appear to be ‘frozen’ until the transfer has completed. For best results, the data transfer functions should be placed in separate threads on the target and host applications. In fact, each direction of transfer should have its own thread, so that the two directions of transfer can interleave as much as possible. The example programs CpuBmIn and CpuBmOut illustrate the use of separate threads for data transfer.

Maximum Transfer Size. The largest transfer allowed is half of the total size of the Dma Buffer allocated by the INF file when the driver is installed. Half of the memory is dedicated to each direction. The default buffer size in the INF is 0x200000 bytes, so the maximum transfer is 1 Megabyte.

Host (Armada Library) Support for CPU Busmastering

The Host *M6713* object contains the following two methods to support CPU Busmastering:

```
//
//  Block Transfer System Methods
```

```
bool       Send(int Channel, const IntBuffer & Block);
bool       Recv(int Channel, IntBuffer & Block);
```

M6713::Send() sends the contents of a *IntBuffer* object to the target**.** All of the data in the *IntBuffer* is transferred. There is no means of sending a partial buffer. The function will not return until the block has been transferred to the target.

The function returns true if the transfer succeeded. It returns false if the transfer failed due to a PCI bus error.

M6713::Recv() waits for data to arrive from the target, then returns the data in the buffer provided. The *IntBuffer* will be re-sized to fit the data transferred from the source. If the buffer is too small, this may involve a reallocation of the data block (which can degrade real-time performance).

The function returns true if the transfer succeeded. It returns false if the transfer failed due to a PCI bus error.

Target (Pismo Library) Support for CPU Busmastering

In the Pismo library, the Utility library contains a file PciTransfer.h that contains this class:

```
class PciTransfer : public PciTransferBase
{
public:
    PciTransfer();

    bool  Send(int channel, const Buffer & buffer);
    bool  Recv(int channel, Buffer & Buffer);
};
```

PciTransfer::Send() sends the contents of a *Buffer*-derived object to the Host**.** All of the data in the buffer is transferred. There is no means of sending a partial buffer. The function will not return until the block has been transferred to the host.

The use of the base buffer class allows any of the *IntBuffer*, *CharBuffer*, *FloatBuffer* and similar classes to be sent across the interface.

The function returns true if the transfer succeeded. It returns false if the transfer failed due to a PCI bus error.

PciTransfer::Recv() waits for data to arrive from the target, then returns the data in the buffer provided. The *Buffer* will be re-sized to fit the data transferred from the source. If the buffer is too small, this may involve a reallocation of the data block.

The function returns true if the transfer succeeded. It returns false if the transfer failed due to a PCI bus error.

See the ASnap example for an illustration of bus-master usage.

C++ Terminal I/O

The terminal emulator applet is a Host PC application which provides a C++ language-compatible, terminal emulation facility for interacting with the TermIo Pismo library running on an Innovative Integration DSP processor.

Using the terminal emulator, it is possible to develop and debug target DSP code while deferring development in Host application code. By using simple, streaming I/O functions within a target application during development, DSP algorithms can be developed independently from Host applications. Later, when a custom Host application code is written, the DSP standard I/O functions may be deleted from the target application and the target application will no longer be dependent on the emulator or the target TermIo libraries.

Streaming methods such as << and >> are dispatched by the TermIo object to route text and data between the DSP target and the Host terminal terminal emulator applet. Text strings are presented to the user via a terminal emulation window and host key-board input data is transmitted back to the DSP. The terminal emulator works almost identically to console-mode terminals common in DOS and Unix systems and provides an excellent means of accessing target program data or providing a simple user interface to control target application operation.

Target Software

All of the features of the terminal are accessed through the two classes `TermIo` and `TermFile`. `TermIo` provides the basic streaming interface which allows text messages to be formatted and streamed out to the terminal as well streaming in strings and numeric values from the terminal for consumption by target application code. The `TermFile` class provides a mechanism allowing target applications to open host disk files, perform read and write accesses, and subsequently close these files. See the `Files.cpp` example for illustratative usage of each of these classes and their functions.

Tutorial

Using the terminal during target software development is simple. The global `cio` object which is automatically instantiated within the Pismo libraries. Use the methods within the `cio` class to format text strings and then stream them to the UniTerminal applet

```
cio << bold << "7Demonstrate file I/Onn" << normal << endl;
```

Note the use of manipulators, such as bold and normal, to force formatting of the text string as it is streamed to the host. TermIo features many such manipulators to perform functions such as setting text color (setcolor), clearing to end-of-line (clreol), clearing the screen (cls) and so forth. Other manipulators are available to format numeric values as they are streamed to the host. For example, the phrase

```
cio << "Hello" << hex << showbase << 4660 << dec;
```

displays the string "Hello 0x1234" on the console display, converting the integer value 4660 as a hexadecimal number on the target, prior to streaming it to the host. Other manipulators are available providing extensive control over the display of floating point numbers as well as integer values.

It is also frequently necessary to obtain input from an operator during the run-time execution of a target application. For example, it may be necessary to prompt for a sample rate at which analog I/O is to be streamed. The code fragment below illustrates the necessary technique

```
// Prompt the user
cio << " Enter a float: " << flush;
float x2;
// Eat user input
cio >> x2;
```

The stream manipulator >> is overloaded to allow streaming directly into floating point, integer and string variables directly from UniTerminal.

To perform file input and output from within target applications, first instantiate a TermFile object, as below

```
TermFile File;
```

Then, use the TermFile Open method to open the file for access on the host using the desired open attributes

```
if (!File.Open("wave.bin", "w+b"))
    {
    cio << "nOutput file open error - Program terminating!" << endl;
    cio.monitor();
    }
```

This method returns a Boolean indicating success if the file open is successful. To store data into the file or retrieve data from the file, use the Write or Read methods, respectively. For example

```
transferred = File.Write((char*)&Buffer[0], 10000);
```

writes 10000 bytes of Buffer into the disk file. When disk operations have been completed, the file should be closed using the TermFile::Close method.

CHAPTER 7

Developing Host Code

This section describes the Innovative Integration Windows host software development environment for the M6713. Support for controlling the board and the network libraries is provided by the native C++ class libraries, named *Malibu,* that can be linked into any application. Sample applications show how to use the Malibu Library to initialize the hardware and communicate with to the hardware in operation.

Malibu for the M6713 is a native C++ class library. It provides a limited amount of software to support threading, buffer manipulation, and other system functions. The use of native C++ code will make the use of the libraries when ported to other platforms straightforward, since the interface for all classes will be identical on all platforms.

Host software development is directly supported under both the Borland C++ Builder 6.0 and Microsoft MSVC v7.0 environments for generating 32-bit Windows applications. While the supplied DataXfer example uses the Borland VCL or MSVC foundation classes for the user interface, the Malibu class library iteself is not dependant on any particular compiler-specific frameworks.

Please Note: Only Windows application development is currently supported by the Developer's Package. Foreign operating systems are not currently supported.

The BoardLib library

All target interactions are implemented bia calls to the supplied Malibu libraries, which are bound into a target executable via supplied static import libraries.

The libraries supply classes that provide the means to operate the M6713 and support classes for common program functions such as threads, buffers, and inter-thread synchronization.

Class Name	Description
M6713	Baseboard support class.
TransferPacketHeader	Command Packet parsing.
MemoryBuffer	Buffer management.
Event	Thread Synchronization.
Thread	Thread class.

Note that this is just a partial listing of available classes within the Malibu libraries. See the online help file, Malibu.chm or Malibu.hlp for details on all available classes, methods and events.

FIGURE 1. Some Classes in Malibu

Board Library Features

Event Handlers. Often in a library it is useful to call user provided code at various points in the operation of the object. In traditional C, pointers to “Callback” functions were often used for this purpose.

Malibu extends on the callback concept through use of *events*. In event-driven programming, user events are a key part of your application logic. An event is a mechanism that links an occurrence to some code. More specifically, an event is a closure that points to a method in a specific class instance.

From the application developer’s perspective, an event is just a name related to a system occurrence, such as OnProgress, to which specific code can be attached. For example, when an error is detected during a system operation, the OnError event might be called.

User events correspond to the elements in your application . For example, during a download operation, the OnDownloadComplete event is called by the system following delivery of the last portion of the executable image to the target DSP. This event can be programmed within application code to update the user interface in an appropriate manner. The code you write to respond to events is called an event handler.

Events and their user-written handlers (called *closures*) are a key part of the Malibu environment. They give us the ability to assign an event handler for system occurances directly within application code.

The M6713 in the Host Environment

The Board Library uses the supplied M6713 device driver for downloading and communication via the PCI bus. The details of this are managed internally by the **M6713** class.

Connection. Each instance of the **M6713** class manages a single board. Systems containing multiple targets are supported and each target is assigned a logical board number starting with zero. After con-

struction, the *Target* method may be used to associate the object with any board installed in the system. Subseqently, a call to the *Open* method establishes communications with that target. Once communications are completed, a call to *Close* discontinues communications.

COFF loading to the 6713 processor. One major requirement is to be able to download and run code on the 6713 over the network. The method *DownloadCoff()* within the *Cpu* data member of the *M6713* object is used to begin a COFF load as a background operation. During the load process, the background thread automatically calls the baseboard class's *OnDownloadProgress* and *OnDownloadComplete* closures to provide interim status. *DownloadCoff()* merely posts a download request into a background thread that actually performs the download of the file passed as as an argument. This method does not block, and will return as soon as the request is posted.

Note that the target application may not be fully initialized and running when *RequestCoffLoad()* is finished, and even after the *OnDownloadComplete* closure is called, since the target initialization routines could be arbitrarily time-consuming . The best approach is to have the host application wait for some message to be sent from the target program which indicates that the target is fully prepared for further communication.

Target / Host Packet Communication. Most applications require some form of contact between the target application and the host application during the running of the program. The M6713 has a bi-directional data channel configured that can be used to send commands and bulk data between the Host CPU and the C6713 DSP.

The communication protocol on this channel supports sending special data blocks consisting of a small header plus an arbitrarily large data packet. These data packets are of type *Innovative::PmcBuffer.* The header within a *PmcBuffer* contains two words of information designated the *PeripheralId* and the *PacketSize*. The *PeripheralId* is an arbitrary tag value stored into the header by the sender intended to allow the receiver to uniquely identify the purpose and contents of the packet. The *PacketSize* field is automatically updated by the driver to accurately reflect the size of the data portion of the packet, in 4-byte words. This field allows the receiver to determine the amount of data payload communicated within the packet.

The *Send()* method transfers both the bulk dataand header stored in an *Innovative::PmcBuffer* object to the target.

Completion of a *Send()* requires that a thread within the target DSP application be blocking within a call to *Transfer::Recv()*. If the target has not done so or the system is otherwise unready, both the sender and receiver will block and not return until the data is completely transfered out and the buffer arguments can be reused. This makes these methods possibly unsafe to call from user-interface code in a Windows application. Since there is some internal queuing in the system, the completion of the send may not mean that the target has actually recieved and processed the packet, either. In general, it is best to call the functions Send and Recv from within background threads only on both Host and Target. However, the Malibu library implements a very efficient event (callback) scheme which can circumvent this requirement in some common cases. See the *ASnap* example for an illustration.

To receive data and commands from the target, use the *Recv()* method. This function will not return until the request is satisfied by receipt of a *PmcBuffer* from the target. The *PeripheralId* field within the header of this buffer may be examined in order to retreive an application-specific code describing the

nature of the data in the packet and the *PacketSize* field contains the size of the data payload contained in the buffer.

The code below illustrates the typical sequence used in the receipt of a buffer from the target:

```
//---------------------------------------------------------------------------
//  TForm1::HandleDataAvailable() --  Handler for all received target packets
//---------------------------------------------------------------------------

void TForm1::HandleDataAvailable(Innovative::PacketStreamDataEvent & Event)
{
    static PmcBuffer Packet;
    static int LoginTally(0);
    //
    // ...Get the packet from the system
    Event.Sender->Recv(Packet);
    PmcDataAccess pda( Packet.Data() );
    //
    // ...Process the packet
    short PacketType = pda.Header()->PeripheralId();
```

In this code, `Packet` is the *PmcBuffer* which will contain the data received from the target. The call to `Event.Sender->Recv(Packet)` blocks until a complete data packet has been received from the target. However, as illustrated in the ASnap example provided with the board, an asynchronous event notification mechanism is provided which provides a callback to a user application funtion such as `HandleDataAvailable` above when data is sent from the target to the host. In that context, the call to the Recv will complete immediately, since data is available.

Once the packet is received, it is common practice to examine the header to discover the type of message sent, then dispatch accordingly. The call to `pda.Header()->PeripheralId()` reads the ID tag sent by the target from the PmcBuffer header. Ordinarily, the application uses a switch table to execute the appropriate code for each ID code that can be received. For example:

```
    switch (PacketType)
        {
        case ccAcqusition:
            {
            Report->Line = "pmAcqusition";
            ProgressBar->StepIt();

            StatusBar->Panels->Items[1]->Text = "Rx: " + String(++RcvMessageTally);

            Cache->Write(pda.IntPtr(), pda.Size()*sizeof(int));
            }
            break;
```

. . .

When the `ccAcqusition` code is received, this application logs the contents of the data payload in the received buffer to disk. The call to `Cache->Write(pda.IntPtr(), pda.Size()*sizeof(int));` calls the Host OS to perform a disk write from the data buffer starting at address `pda.IntPtr()` which is the address of the first byte of the data payload within the packet, cast as an integer pointer. The number of bytes in the payload are returned by the phrase `pda.Size()*sizeof(int)`.

The target software uses a similar method for the other end of the connection. The **Transfer** class has identical *Send()*, and *Recv()* methods of its own for the same purposes.

Host Example Program for the M6713 Baseboard

Overview

Command and Data Packets. On this baseboard, using PCI bus, the way to transport information is by means of asynchronous packets of data that are transferred and decoded by the destination. These messages may be of varying sizes, allowing large messages to be efficient in sending bulk data while allowing small command messages to be mixed into the data stream. When delivered to the destination, the messages can be parsed and can result in any kind of processing desired.

By having the receipt of a message trigger the generation of additional messages, a message protocol can be developed to allow the transfer of data or the execution of control functions on demand from the other side of the link. This is even more natural if the applications are written in an event-driven style. The arrival of messages are the events to which the application responds.

ASnap

The ASnap example is communications demonstration that resides in the \M6713\Examples\ASnap directory. This example demonstrates continuous, high-rate analog acquisition from a suitable Omnibus module into a buffer on the target DSP. After the acquisition, the data is sent to a Windows disk fle via the PCI interface. This data may be examined using the supplied BinView applet. Simple bidirectional communications between the target and the host are shown.

The target project and its source are located in the \M6713\Examples\Asnap directory. However, there are two versions of the Host project and source, located in the \BCB and \Vc subdirectories beneath the target project. These contain the source for the Borland BCB and Microsoft MSVC versions of the Host example, respectively.

CHAPTER 8

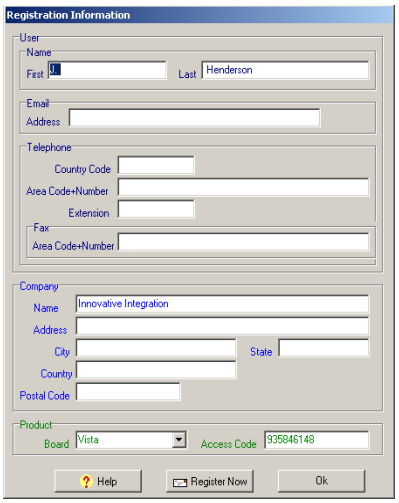
Applets

This chapter describes the Host PC utility applets that are provided with the Pismo tool suite.

To invoke any of these utilities, go to the Start menu | Programs | M6713 menu and click the one you are interested in running.

Registration Utility (NewUser.exe)

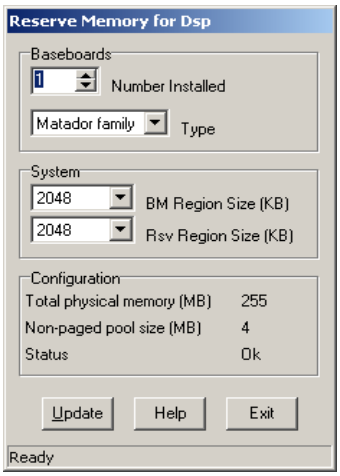
Some of the Host applets provided in the Malibu Developers Package are keyed to prevent unauthorized duplication. These utilities allow unrestricted use for up to 20 days (trial period), during which you are required to register your Toolset. After the trial period, operation will be disallowed until an unlock code, provided by Innovative Integrationm is used to re-enable the applet. After using the NewUser.exe applet to provide Innovative Integration with your registration information, you will receive:



- The unlock code necessary for unrestructed use of the Host applets
- A WSC (tech-support service code) enabling free software maintenance downloads of development kit software and telephone technical hotline support for a one year period.

ReserveMemoryDsp

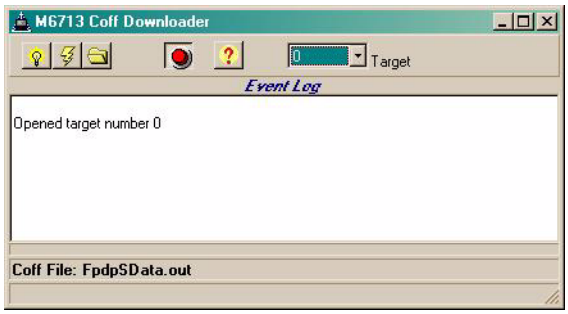
Each Innovative PCI-based DSP baseboard requires from 2 to 8 MB of memory to be reserved for its use, depending on the rates of bus-master transfer traffic which each baseboard will generate. Applications operating at transfer rates in excess of 20 MB/sec should reserve additional, contiguous busmaster memory to ensure gap-free data acquisition.



To reserve this memory, the registry must be updated using the ReserveMemDsp applet. If at any time you change the number of or rearrange the baseboards in your system, then you must invoke this applet found in Start | Programs | Matador | ReserveMemoryDsp. See the help file, ReserveMemDsp.hlp, for operational details.

Target Download Utility (M6713Download.exe)

The download applet is used to deliver known-operational DSP executables to DSP baseboards. The utility may be used to start DSP applications on PC power-up, through its command line interface, or to start a DSP application from its GUI Windows user interface. It is also capable of downloading a minimal "boot" application, which is convienient when attempting to start a new Code Composer debug session after having initialized the JTAG scan path with JtagDiag.exe.



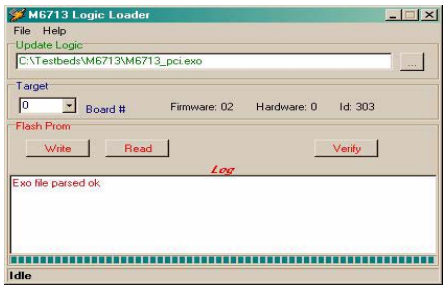
Logic Download Utility (M6713LogicLoader.exe)

The logic download applet is used to deliver known-operational logic images to either of the logic devices installed on an M6713 baseboard. The utility may be used to configure firmware either through its command line interface or from its GUI Windows user interface. The former is often convenient during PC boot-up.

This application supports configuration of the onboard Spartan3 logic device from an EXO file produced by popular logic design tools (including Xilinx's). It is essential that the Spartan be programmed before attempting to download COFF images to the DSP, since some of the baseboard peripherals are dependent on the personality of the configured logic.

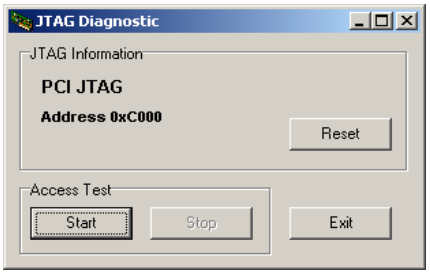
Logic Update Utility (M6713VsProm.exe)

The Logic Update Utility applet is designed to allow field-upgrades of the logic firmware on Modular baseboards. The utility permits an embedded firmware logic update file to reprogrammed into the baseboard Flash ROM, which stores the "personality" of the board. Complete functionality is supplied in the application's help file.



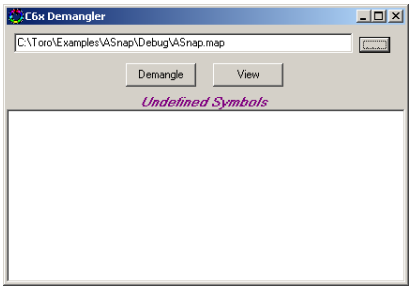
JTAG Diagnostic Utility (JtagDiag.exe)

JtagDiag.exe is used to re-initialized the JTAG scan-path interface which connects the Code Hammer debugger's PCI plug-in board with the target DSP. Use this utility prior to invoking Code Composer Studio, to insure that the communications link is viable and clear. This utility is also convienient in confirming that the Code Hammer installation is complete and correct.



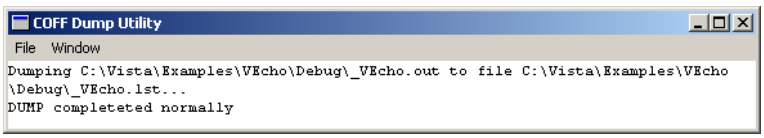
Demangle Utility (Demangle.exe)

The Demangle applet is designed to simpify use of the TI dem6x.exe command-line utility. When building C++ applications, the built-in symbol mangler in the TI compiler renders symbolic names unreadable, such that missing or unresolved symbol errors displayed by the linker no longer correlate to the symbol names within your code. To work around this limitation, enable map file generation within your CCS project. Then, browse to the map file produced by the linker using the Demangle utility. The utility will display proper symbol names for all unresolved externals.



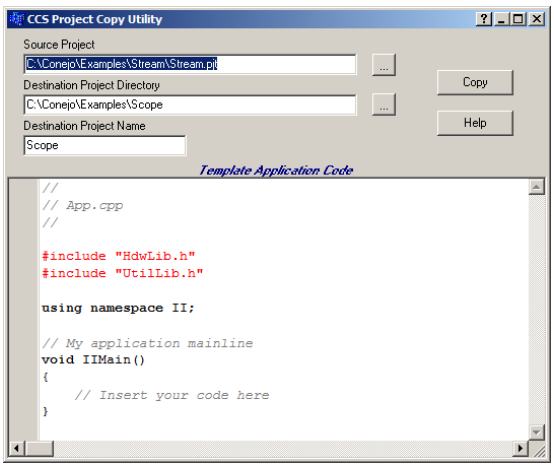
COFF Section Dump Utility (CoffDump.exe)

CoffDume.exe parses through a user-selected COFF file stored on the hard disk and ascertains the complete memory consumption by the DSP program. Memory usage for each of the sections defined in the applications command file are tabularized and the results are written to the Windows NotePad scratch buffer.



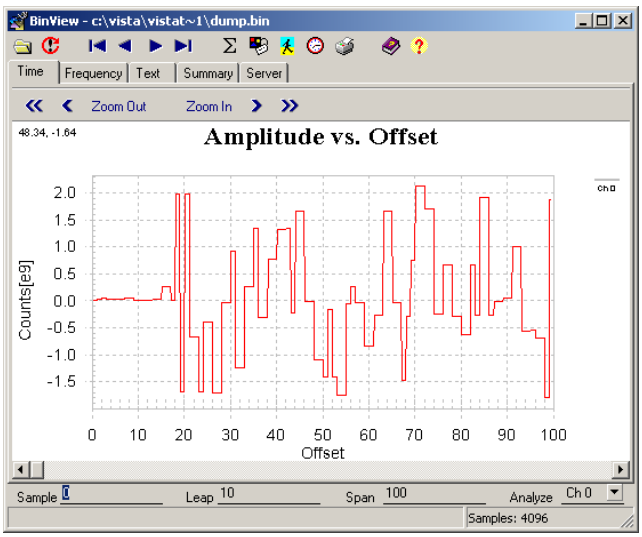
Target Project Copy Utility (CopyCcsProject.exe)

The `CopyCcsProject.exe` applet is used to copy all project settings from a known-good template project into a new DSP Code Composer project. This simplifies new project development, by eliminating the multi-step process of copying the myriad individual project settings from a source project in a newly-created project.



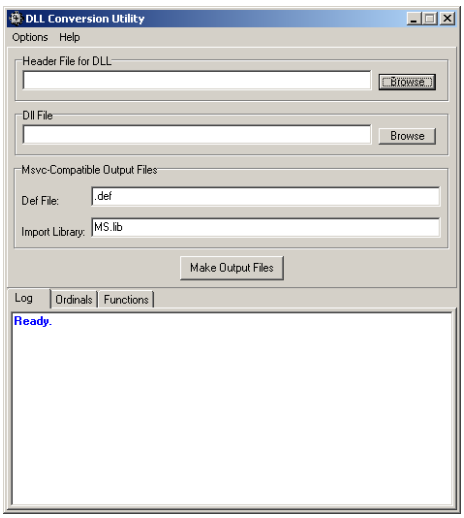
Binary File Viewer Utility (BinView.exe)

BinView is a data display tool specifically designed to allow simplified viewing of binary data stored in data files or resident in shared DSP memory. Please see the on-line BinView help file in your Binview installation directory.



DEF Conversion Utility (DefConvert.exe)

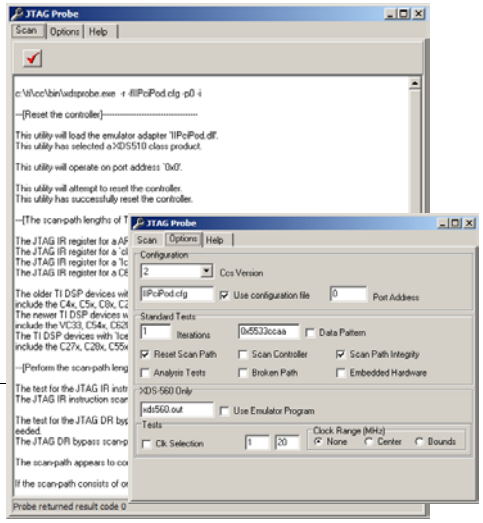
The DefConvert.exe applet is a simple utility to aid in the use of DLLs written using Borland Builder from within Microsoft Visual C/C++- application programs. DefConvert parses a user specified C++ header file and DLL in order to create a MSVC compatible DEF file. This DEF file is then submitted to the MSVC LIB utility in order to generate an MSVC-compatible import library (.LIB).



Complete functionality is supplied in the DefConvert.hlp file.

Scan Path Diagnostic Utility (JtagScanpath.exe)

The JtagScanpath.exe applet is a simple GUI front-end to the powerful Texas Instruments XdsProbe.exe command-line utility. The utility is of value in debugging JTAG debugger installation and reliability problems. The tool is capable of performing comprehensive scan integrity tests for both the Innovative Code Hammer and the TI XDS560 emulators. The complete reference to available tests and features is listed on the application Help tab.



RtdxTerminal "The Terminal Emulator"

This applet provides a C++ language-compatible, standard I/O terminal emulation facility for interacting with the *TermIo* library running on an Innovative Integration target DSP processor. Display data is routed between the DSP target and this Host the terminal emulator applet in which ASCII output data is presented to the user via a terminal emulation window and host keyboard input data is transmitted back to the DSP. The terminal emulator works almost identically to console-mode terminals common in DOS and Unix systems, and provides an excellent means of accessing target program data or providing a simple user interface to control target application operation during initial debugging.

RtdxTerminal is implemented as an out-of-process extension to Code Composer Studio. Consequently, it must be used in conjunction with CCS and a JTAG debugger - it cannot operate stand-alone.

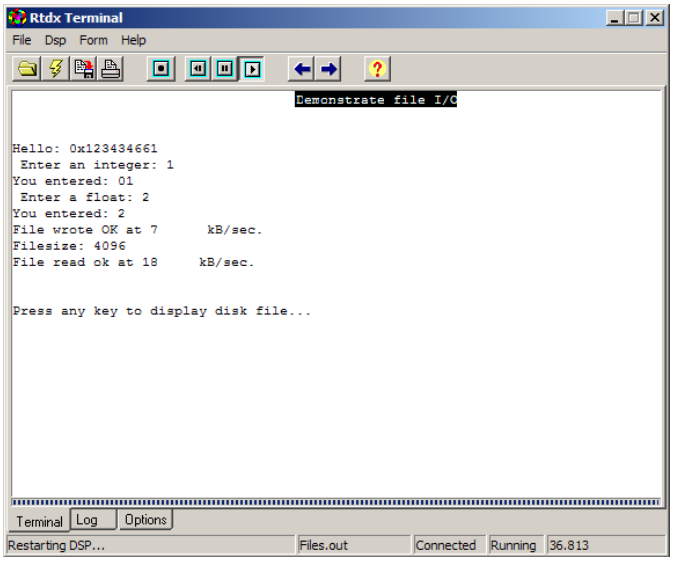


FIGURE 2. Terminal Emulator Applet

The terminal emulator is straightforward to use. The terminal emulator will respond to stdio calls automatically from the target DSP card and should be running before the DSP application is executed in order for the program run to proceed normally. The DSP program execution will be halted automatically at the first stdio library call if the terminal emulator is not executing when the DSP application is run, since standard I/O uses hardware handshaking. The stdio output is automatically printed to the current cursor location (with wraparound and scrolling), and console keyboard input will also be displayed as it is echoed back from the target.

The terminal emulator also supports Windows file I/O using the TermFile library object.

Important Note: Before using the terminal emulator, you must register your Pismo Toolset. Until you do so, usage will be restricted to a 20-day trial period for the terminal emulator and other applets contained in the Toolset. To register, fill out the contents of the Registration Form, then click on the Register Now button. This will print a Registration report which, must be faxed to Innovative Integration. Innovative Integration will E-mail you an Access Code, which must be typed into the Registration Form for all the features to be enabled.

Terminal Emulator Menu Commands. The terminal emulator provides several menus of commands for controlling and customizing its functionality. These functions are available on the menu bar, located at the top of the the terminal emulator main window. Speed button equivalents for each of the menu options are also available on the button bar located immediately beneath the menu bar. The following is a description of each menu entry available in the terminal emulator, and its effects.

File Menu:

.

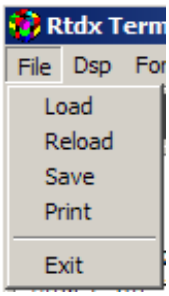


FIGURE 3. RtdxTerminal File Menu

- File | Load - provides for COFF (Common Object File Format) program downloads from within the terminal emulator. When selected, a file requester dialog box is opened and the full pathname to the COFF filename to be downloaded is selected by the user. Clicking “Open” in the file requester once a filename has been selected will cause the requester to close and the file to be downloaded to the target and executed. Clicking “Cancel” will abort the file selection and close the requester with no download taking place.

  This operation can optionally be initiated via the button.

- File | Reload - Reloads and executes the COFF file last downloaded to the target. It provides a fast means to re-execute the application program most recently loaded into the target board.

  This operation can optionally be initiated via the button.

NOTE: File | Load and File | Reload functions use the JTAG debugger and Code Composer Studio in order to effect the program download.

- File | Save – saves the textual contents of the Terminal and Log tabs to a user specified file.

- File | Print - prints the textual contents of the Terminal and Log tabs to a user specified printer.

- File | Exit – closes the emulator application, terminating console emulation.

DSP Menu:

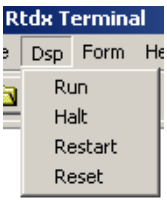


FIGURE 4. RtdxTerminal DSP Menu

- Dsp | Run - causes the terminal emulator to bring the target board into a cold-start, uninitialized condition. This is functionaly identical to performing Debug | Run within Code Composer Studio.

  This operation can optionally be initiated via the button.

- Dsp | Halt - causes the terminal emulator to suspend DSP program execution. This is functionaly identical to performing Debug | Halt within Code Composer Studio.

  This operation can optionally be initiated via the button.

- Dsp | Restart - rewinds the DSP program counter to the application entry point, usually `c_int00()`. This is functionaly identical to performing Debug | Restart within Code Composer Studio.

  This operation can optionally be initiated via the button.

- Dsp | Reset - causes the terminal emulator to bring the target board into a cold-start, uninitialized condition. This is functionaly identical to performing Debug | Reset Dsp within Code Composer Studio.

  This operation can optionally be initiated via the button.

Form Menu:

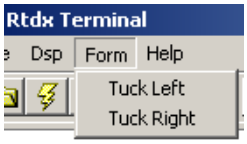


FIGURE 5. RtdxTerminal Form Menu

- Form | Tuck Left - repositions the main application window to the bottom left of the Windows desktop.

  This operation can optionally be initiated via the button.

- Form | Tuck Right - repositions the main application window to the bottom right of the Windows desktop.

  This operation can optionally be initiated via the button.

Help Menu:

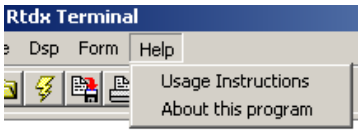


FIGURE 6. RtdxTerminal Help Menu

- Help | Usage Instructions - displays online help detailing use of the application, including command-line arguments.

  This operation can optionally be initiated via the button.

- Help | About this Program - displays a dialog containing program revision and tech support contact information.

Options Tab:

The Options tab (seen below) contains controls to allow user-customization of the appearance and operation of the terminal emulator.

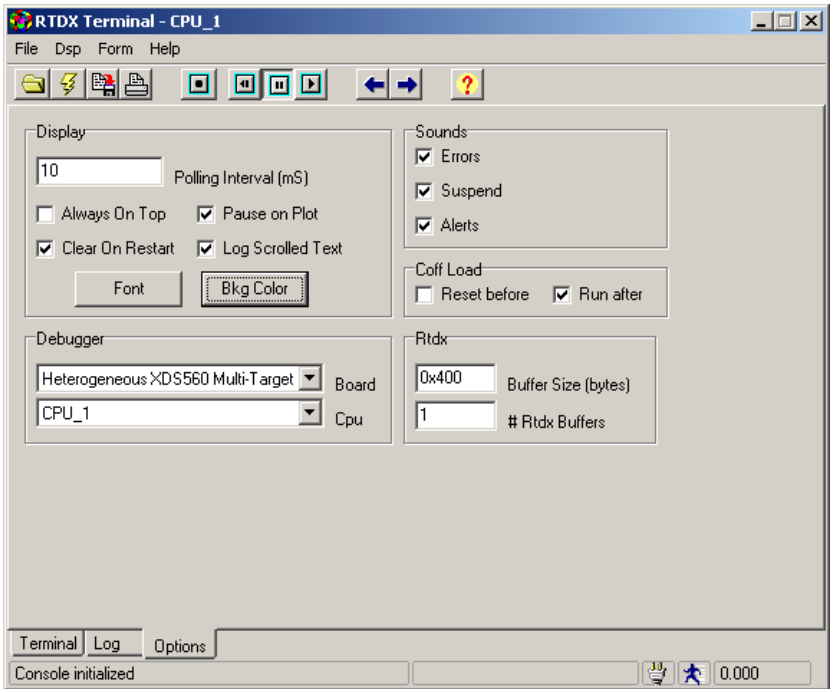


FIGURE 7. RtdxTerminal Options

Display Group

Controls within the Display group box govern the visual appearance of the terminal emulator, as detailed below.

- Polling Interval - specifies the period, in milliseconds, between queries for data received from the DSP via the JTAG RTDX interface. Lower numbers increase performance but increase Host CPU load.
- Always on Top - specifies that the terminal application should always remain visible, atop other applications on the Windows desktop. This check box controls whether the terminal emulator is forced to remain a foreground application, even when it loses keyboard focus. This is useful when running stdio-based code from within the Code Composer environment, when it's preferable to make terminal visible at all times. The terminal will remain atop other windows when this entry is checked. Select the entry again to uncheck and allow the terminal emulator window to be obscured by other windows.
- Clear on Restart - specifies whether the terminal display and log will be automatically cleared whenever the DSP is restarted.
- Pause on Plot - specifies whether standard I/O will be suspended following display of graphical information in the Binview applet which is automatically invoked via use of the Pismo library `Plot()` command. If enabled, standard I/O may be resumed by clicking the button.
- Log Scrolled Text - specifies whether text information which scrolls offscreen on the Terminal tab is appended to the Log display. If enabled, standard I/O performance will degrade slightly during lengthy text outputs.
- Font - button invokes a font-selection dialog which allows selection of user-specified font within the Terminal and Log text controls.

- Bkg Color - button invokes a color-selection dialog which allows selection of user-specified background color within the Terminal and Log text controls.

Sounds Group

Controls within the Sounds group box govern the audible prompts generated by the terminal emulator, as detailed below.

- Errors - if enabled, file I/O and other errors encountered during operation generate an audible tone.
- Suspend - if enabled, suspension of standard I/O, such as following plotting via Binview, generate an audible tone.
- Alerts - if enabled, alert conditions encountered during standard I/O, such as upon display of the ASCII bell character, generate an audible tone.

Coff Load Group

Controls within the Coff Load group box govern behaviour surrounding a COFF executable download.

- Reset Before - if enabled, the Code Composer Debug | Reset DSP behaviour is executed before attempting to download the user-specified COFF file.

Run After - if enabled, the Code Composer Debug | Run behaviour is executed immediately following the download of a user-specified COFF file.

Debugger Group

Controls within the Debugger group box specify the target DSP with which RTDX communications is established.

- Board - specifies the board hosting the target DSP to be used in RtdxTerminal stdio communications. This combo box is populated with all available board types configured using the Code Composer Setup utility.
- Cpu - specifies the identifier of the specific DSP to be used in RtdxTerminal stdio communications. This combo box is populated with all available CPUs present on the baseboard as configured using the Code Composer Setup utility.

Terminal Emulator Command Line Switches. The terminal emulator also provides the following command line switches to further modify program behavior. The switches must be supplied via the command line or within Windows shortcut properties (see the Installation section for more information), and will override the default behavior of the applet.

Multiple instances of the terminal emulator may be invoked simultaneously in order to support installations utilizing multiple target boards. Instances of the terminal emulator, after the first loaded instance must be configured via command line switches in order to properly communicate with their associated target.

- **-board** *boardtype* - Use the -board switch to force an instance of the terminal emulator to communicate with a specific type of target board, boardtype. Supported boardtypes are those configured using the Code Composer Setup utility, such as “C64xx Rev 1.1 XDS560 Emulator”.

- **-cpu** *cputype* - Use the -cpu switch to force an instance of the terminal emulator to communicate with a specific CPU on a target board. Supported CPU types are those configured using the Code Composer Setup utility, such as “CPU_1” or “CPU_A”.

- **-f** *filespec* - Use the -f switch to force the terminal emulator to load and run the specified COFF file. The *“filespec”* field should be a standard Windows file specification, including both the path and file name as a unit, to allow the user to force the terminal emulator to download the specified file to the target DSP board, as soon as the terminal emulator is loaded. This field is particularly useful in situations where the the terminal emulator is “shelled to” from within an other Host applications to facilitate the automatic execution of target applications employing standard I/O.

CHAPTER 9

M6713 Hardware

M6713 Hardware Functions

The M6713 is a PCI bus plug-in digital signal processor (DSP) card based around the Texas Instruments TMS320C6713 processor. The M6713 is particularly well suited to data acquisition and control tasks and may be equipped with a wide range of Omnibus IO modules for analog and digital interfaces. A high performance PCI bus link provides data and control connectivity for network-based instrumentation and test equipment.

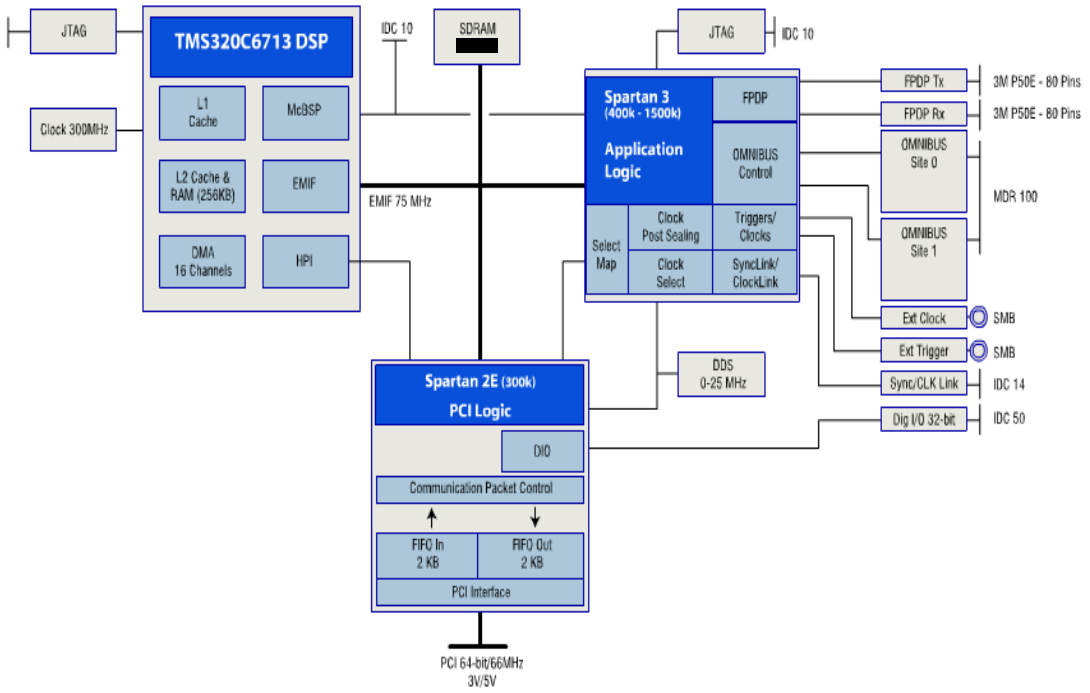
The M6713's features include:

1. TMS320C6713 Floating point digital signal processor.
2. 128MB SDRAM .
3. Xilinx Spartan3 1.5M gate FPGA programmable using MATLAB Simulink and VHDL.
4. 200 MB/sec full duplex FPDP communications expansion capability.
5. Analog and digital I/O expansion using Omnibus compatible I/O modules (two available slots).
6. 32 bits of digital I/O.
7. Synclink and clocklink trigger and clock sharing bus.
8. High Performance PCI bus controller.

9. JTAG hardware emulation support.

The following figure gives a block diagram of the M6713.

FIGURE 8. M6713 Block Diagram



Memory Map

There are two memory maps for the M6713: the DSP memory map and the PCI-mapped devices on the card.

The following figure gives the DSP memory map of the M6713 for external peripherals and memory. Please note that this table ignores any on-chip resources (see TI peripheral manual for TMS320C6713 for on-chip peripherals).

CE Space	DSP Memory Address	Logic Device	Logic Address (Decimal)	Access Type	Read/ Write	Description
Ce0	0x80000000	PCI	0	Async	W	DIG_config
Ce0	0x80010000	PCI	1	Async	W	DIG_data

Ce0	0x80020000	PCI	2	Async	W	DDS control
Ce0	0x80030000	PCI	3	Async	W	DDS data
Ce0	0x80040000	PCI	4	Async	W	
Ce0	0x80050000	PCI	5	Async	W	INT4 status/ack
Ce0	0x80060000	PCI	6	Async	W	
Ce0	0x80070000	PCI	7	Async	W	DMA int enable
Ce0	0x80080000	PCI	8	Async	W	NMI status/ack
Ce0	0x80090000	PCI	9	Async	W	INT4 enable
Ce0	0x800A0000	PCI	10	Async	W	INT5 enable
Ce0	0x800B0000	PCI	11	Async	W	INT6 enable
Ce0	0x800C0000	PCI	12	Async	W	INT7 enable
Ce0	0x800D0000	PCI	13	Async	W	NMI enable
Ce0	0x800E0000	PCI	14	Async	W	Not used
Ce0	0x800F0000	PCI	15	Async	W	Not used
Ce0	0x80100000	PCI	16	Async	W	INT type
Ce0	0x80110000	PCI	17	Async	W	INT polarity
Ce0	0x80120000	PCI	18	Async	W	Burst count register – INT4
Ce0	0x80130000	PCI	19	Async	W	Burst count register – INT5
Ce0	0x80140000	PCI	20	Async	W	Burst count register – INT6
Ce0	0x80150000	PCI	21	Async	W	Burst count register – INT7
Ce0	0x80160000	PCI	22	Async	W	Burst count register – DMA int
Ce0	0x80170000	PCI	23	Async	W	Not used
Ce0	0x80180000	PCI	24	Async	W	PCI FIFO level control register
Ce0	0x80190000	PCI	25	Async	W	Not used
Ce0	0x801A0000	PCI	26	Async	W	Not used
Ce0	0x801B0000	PCI	27	Async	W	Burst address register – INT4
Ce0	0x801C0000	PCI	28	Async	W	Burst address register – INT5
Ce0	0x801D0000	PCI	29	Async	W	Burst address register – INT6
Ce0	0x801E0000	PCI	30	Async	W	Burst address register – INT7
Ce0	0x801F0000	PCI	31	Async	W	Burst address register – DMA int

Ce0	0x80200000	Intf	0	Async	W	Control reg
Ce0	0x80200004	Intf	1	Async	W	SyncLink config
Ce0	0x80200008	Intf	2	Async	W	ClockLink config
Ce0	0x8020000C	Intf	3	Async	W	Module site trigger config
Ce0	0x80200010	Intf	4	Async	W	FPDP Rx BITIO
Ce0	0x80200014	Intf	5	Async	W	FPDP Rx Config
Ce0	0x80200018	Intf	6	Async	W	FPDP Tx Config
Ce0	0x8020001C	Intf	7	Async	W	FPDP Tx Frame Count
Ce0	0x80200020	Intf	8	Async	W	Selection Synclink/Counters Register
Ce0	0x80200024	Intf	9	Async	W	Module 0 timebase selections
Ce0	0x80200028	Intf	10	Async	W	Module 1 timebase selections
Ce0	0x8020002C	Intf	11	Async	W	DDS0 post-scaling register
Ce0	0x80200030	Intf	12	Async	W	DDS1 post-scaling register
Ce0	0x80200034	Intf	13	Async	W	Timer 0 Configuration Write
Ce0	0x80200038	Intf	14	Async	W	Timer 1 Configuration Write
Ce0	0x8020003C	Intf	15	Async	W	Timer 2 Configuration Write
Ce0	0x80200040	Intf	16	Async	W	Timer Write 0 (Counter End Register)
Ce0	0x80200044	Intf	17	Async	W	Timer Write 1 (Counter End Register)
Ce0	0x80200048	Intf	18	Async	W	Timer Write 2 (Counter End Register)
Ce0	0x80200040	Intf	16	Async	R	Timer 0 counter value read
Ce0	0x80200044	Intf	17	Async	R	Timer 1 counter value read
Ce0	0x80200048	Intf	18	Async	R	Timer 2 counter value read
Ce0	0x8020004C ..0x8020005C	Intf	19..23	Async	R/W	Not used
Ce0	0x80200060	Intf	24	Async	R	Status Register Read
Ce0	0x80200064	Intf	25	Async	R/W	Not Used
Ce0	0x80200068	Intf	26	Async	R/W	FPDP Rx PIO

Ce0	0x8020006C	Intf	27	Async	R	FPDP Rx Frame Count
Ce0	0x80200070	Intf	28	Async	R/W	FPDP Tx PIO
Ce0	0x80200074	Intf	29	Async	R	FPDP Tx BITIO
Ce0	0x80200078	Intf	30	Async	R/W	Not Used
Ce0	0x8020007C	Intf	31	Async	R/W	Not Used
Ce1	0x90000000	PCI	0	Burst	R	PCI fifo data read
Ce1	0x90000000	PCI	0	Burst	W	PCI fifo data write
Ce1	0x90200000	Intf	0	Burst	R	FPDP Rx fifo data
Ce1	0x90200000	Intf	0	Burst	W	FPDP Tx fifo data
Ce2	0xA0000000	Not connected to logic		Burst	R/W	SDRAM
Ce3	0xB0000000	Intf	0	Async	R/W	IOMOD0
Ce3	0xB0004000	Intf	1	Async	R/W	IOMOD1
Ce3	0xB0008000	Intf	2	Async	R/W	IOMOD2
Ce3	0xB000C000	Intf	3	Async	R/W	IOMOD3
Ce3	0xB0010000	Intf	4	Async	R/W	IOMOD4
Ce3	0xB0014000	Intf	5	Async	R/W	IOMOD5
Ce3	0xB0018000	Intf	6	Async	R/W	IOMOD6
Ce3	0xB001C000	Intf	7	Async	R/W	IOMOD7

TABLE 13. M6713 DSP External Memory Map

M6713 Hardware Initialization Requirements

The M6713 design requires the following values to be written to its hardware control registers in order to provide access to on-board hardware:

Register	Address	Value
EMIF Global Control	0x01800000	0x00003078
CE1 Control	0x01800004	0x0000C041
CE0 Control	0x01800008	0x21A28A22
CE2 Control	0x01800010	0x00000030
CE3 Control	0x01800014	0x11010420
SDRAM Control	0x01800018	0x6B338000
SDRAM Refresh	0x0180001C	0x00000350
SDRAM Extension	0x0180001C	0x000544a7
Interrupt Polarity	0x019C0008	0x00000000

TABLE 14. M6713 Bus Control Register Initialization Values

These values are initialized automatically by C++ programs compiled under the M6713 Development Package software libraries. Be sure to include initialization of these values whenever software is developed outside the Development Package or when a JTAG hardware assisted debugger is employed for code downloading to the M6713 (i.e. when using Code Composer Studio or any other JTAG debugger package).

The DSP also must initialize the clock PLL and steering. The input clock is 37.5MHz, which is multiplied by 8 in the DSP PLL for the main processor 300 MHz clock. The clock control in the DSP is also configured to provide a 75 MHz clock on SYSCLK3 for the EMIF clock. Here are the intialization values as used in the support software.

For 300 MHz Processor

Divider D0:	/1	(PLLREF = 37.5 MHz)
PLL:	x8	(PLLOUT = 300 MHz)
Divider D1:	/1	(SYSCLK1 = 300 MHz)
Divider D2:	/2	(SYSCLK2 = 150 MHz, must be ½ SYSCLK1)
Divider D3:	/4	(SYSCLK3 = 75 MHz, for EMIF and so that Omnibus is 37.5MHz)

External Memory

The M6713 external memory is synchronous DRAM (SDRAM) organized as 32Mx32 (128 Mbytes). The SDRAM operates at a fixed rate of 75 MHz, for a maximum burst throughput of 300 Mbytes/sec.

Practical use of the SDRAM for the best performance normally requires thoughtful allocation of the DSP on-chip memory for local memory and cache use. SDRAM is only fast to access in bursts to consective memory addresses; random accesses to memory are very slow, often requiring at least 6 external clock cycles per access. The cache controller .

The ‘C6713 has an advanced cache controller that helps access times by putting data into local DSP memory from the SDRAM. This makes the SDRAM look to the programmer like a large virtual memory pool of very fast memory. The cache controller allows the SDRAM to operate at approximately

80% the efficiency of internal memory in many cases by caching instructions and data in on-chip memory for faster access.

M6713 OMNIBUS

The M6713 has two Omnibus I/O sites mapped into the processor memory space. A variety of Omnibus IO modules are available with many types of analog and digital IO that can be mixed and matched to suit the particular user's functional requirements. Omnibus is an open standard that allows customers to design application specific modules to use with the M6713 and other Omnibus cards.

The OMNIBUS slots are accessed as memory-mapped peripherals with the M6713 providing four decoded chip select signals per slot, for a total of eight on the M6713. The following figure gives the memory map for the OMNIBUS slots, and shows the decode signal to slot mapping.

Function	Starting Address	Module Slot
OMNIBUS Strobe 0	0xB0000000	0
OMNIBUS Strobe 1	0xB0010000	0
OMNIBUS Strobe 2	0xB0020000	0
OMNIBUS Strobe 3	0xB0030000	0
OMNIBUS Strobe 4	0xB0040000	1
OMNIBUS Strobe 5	0xB0050000	1
OMNIBUS Strobe 6	0xB0060000	1
OMNIBUS Strobe 7	0xB0070000	1

TABLE 15. M6713 I/O Bus Memory Mapping

Each module site provides a 32-bit wide data bus connection to the processor's data bus, with 12-bits of low order address signals for additional decoding beyond the four chip select signals available per slot. Since the Omnibus modules are 32-bit devices, and the DSP has byte addressing, the address mapping to the M6713 is such that Omnibus address A0 is address A2 on the DSP.

Each module also connects to a 'C6713 serial port (serial port zero for slot zero, and serial port 1 for slots 1) to allow serial port driven I/O. Bus reset, RDY, R/W, and processor clock signals are available, as are power connections for digital 5V and analog +/-5V and +/-15V. Timebase connections include timer channels from both the 16-bit timers and the AD9851 direct-digital synthesizer (DDS).

Each OMNIBUS slot has a 50 pin undedicated connector (JP3 on slot 0 and JP7 on slot 1) that provides access from the external I/O to and from a module installed in the slot. The slot's I/O connector is in turn pinned out to a single 100 pin MDR connector (JP4) for use in attaching cables from external hardware. Cables and breakout modules for the MDR 100 output connector are available from Innovative. Custom cables may also be made using the connectors specified in the appendix.

Connector pinouts for the module sites are provided in the appendices. Individual pin functions are noted in the tables, and in general the OMNIBUS pinout represents a direct connection to the 'C6713 local bus.

M6713 OMNIBUS Memory Mapping

Since the ‘C6713 processor is a byte addressable machine which implements its address bus based on a 32-bit transfer width (i.e. the address bus starts at A2 and separate byte enable pins are supplied to control accesses to individual bytes within the 32-bit wide location denoted by the address bus), users must take care when writing software which performs OMNIBUS accesses.

The OMNIBUS specification requires 32-bit accesses and does not support byte or half-word (16-bit) accesses. No support is included in the specification for the ‘C6713’s byte enable pins. This means that software performing accesses must always perform 32-bit transactions with the OMNIBUS modules. When writing C code for the M6713, programmers should use only variables of type int or unsigned int (or their derived types), and all accesses should be word justified (the least significant nibble of the address must always be a multiple of four). Accesses generated using pointers to variables of type char, short, or long will cause erroneous non-32-bit accesses. Correct OMNIBUS module operation under these situations can not be guaranteed.

Please note that memory decoding within the OMNIBUS decode regions uses 32-bit addressing and that the memory map tables given in the *OMNIBUS Manual* should be treated appropriately. For example, the description of the OMNIBUS DIG module notes that the byte 3 direction control register for a module installed in site 0 is mapped to address IOMOD2 + 3. This address should be literally interpreted as 0xB002000C, where IOMOD2 is equal to 0xB0020000 and the offset adds decimal 12 (three 32-bit words of offset). IOMOD2 + 3 should NOT be interpreted as 0xB002003, since the offset is 3 32-bit words and not 3 bytes.

This addressing is most easily handled in C by using integer pointers and integer pointer arithmetic, which will always result in the required address alignment. For example, the following code defines a pointer and accesses the byte 3 direction control register with the documented offset:

```
unsigned int *pointer = 0xB0020000;

*(pointer + 3) = 0x0;       /* set byte 3 to output mode */
```

The actual accessed memory location is 0xB002000C, due to the way pointer math is handled in C.

OMNIBUS Power

The OMNIBUS interface provides six separate power supplies for use by modules along with two separate ground return connections. The following table lists the power supplies and their power ratings. A separate digital 5V supply is provided along with separate digital grounds to minimize the digital noise present on the analog power supplies.

Pin Name	Voltage	Current Rating (max.)
DVCC	5V (digital)	(System dependent)
AVCC	5V (analog)	500 mA
-AVCC	-5V	500 mA
+AV	+15V	(System dependent)
-AV	-15V	(System dependent)
3.3V	+3.3V	250 mA

TABLE 16. I/O Bus Power Ratings

Note: The M6713 implementation of the OMNIBUS expansion slots deviates from the standard specification in the way the power supplies are handled. In order to provide a more compact form factor for the host card, the M6713 does not supply discrete +/-12V power supplies as required by the OMNIBUS specification. Instead, it connects the +/-AV rails to the +/-12V OMNIBUS power pins, resulting in a 3V absolute overvoltage on those supply pins. Designers of custom OMNIBUS modules intended for use with the M6713 should keep these revised power supply values in mind when planning circuitry, which connects to these power supplies.

Please note that the AGND and DGND busses are separated on the M6713 and for proper ground referencing they must be tied together on modules which use the analog power supplies (any supply other than digital 5V, 12V, or –12V). Innovative Integration recommends that either a ferrite bead (Panasonic EXC-ELSA35V or equivalent) or a hard wire connection, depending on expected ground return current and frequency content from the analog power supplies being used on custom modules to connect the two ground busses. The current steering used by the separate grounds prevents high frequency digital noise on the DGND bus from polluting the clean AGND return.

Omnibus Data Rates

Omnibus data rates vary from module to module. Most modules can achieve a data rate of 48MB/s on the M6713 as designed. The Omnibus clock is 37.5 MHz on the M6713 so the typical module access is 3 clocks.

Custom designs where the Omnibus interface is replaced in the baseboard and module logic can achieve data rates of up to 200 MB/s. For these rates, custom logic must be implemented that supports synchronous data transfers top the M6713.

Designing Custom Omnibus Modules

Custom Omnibus designers should review the Omnibus Specification from Innovative. We also mechanical drawings to assist customers in design. Logic designs showing typical interfaces with FIFOs and memory decoding for FPGAs are also available from Innovative. Contact technical support for this information.

FPDP Port I/O Expansion

The Front Panel Data Port (FPDP) feature provides two 32-bit data ports, one for input and one for output, used for communicating with other I/O devices or DSP cards. These ports support a FPDP as defined in VITA 17 specification. The FPDP bus is intended to provide data transfer between two or more VMEbus (i.e. Versa Module Europa Bus) boards up to 200MB/s with the lowest possible latency. FPDP is a 32-bit parallel synchronous bus wired by means of an 80-conductor ribbon cable connector at the front of the VMEbus board. A single master generates a free-running clock (Data Strobe or +/- PECL Data Strobe), the frequency of which defines the maximum transfer rate of the bus. The bus protocol does not include address or arbitration cycles, so the data transfer rate is fully defined by the fre-

quency of the Data Strobe. A mechanism is provided to allow a receiver to hold off the transmitter if its memory is almost full; this is done using the *Suspend Data Signal*. A mechanism is also provided to allow synchronization of the receiver to the transmitter data stream to provide for memory initialization and so that framed data can be correctly interpreted; this is done using the *Sync Pulse signal*.

In the Interface FPDP Logic, we have provided 512x32 FIFO memory in each direction (i.e. Transmitter FIFO and Receiver FIFO) . Data is deposited in Tx FIFO port by the DSP as demanded by the process, on the other hand Rx FIFO port is read by the DSP as it receives the data from other I/O devices. An interrupt may be signalled to the DSP based on a programmable FIFO level, referred to as the interrupt threshold level. For data acquisition applications, larger packets may be used to reduce the DSP interrupt rate at the expense of data latency. DMA or CPU transfers MUST consume the same number of points as the threshold value before another interrupt will be signalled.This prevents spurious interrupts as the FIFO crosses the threshold value during reads.

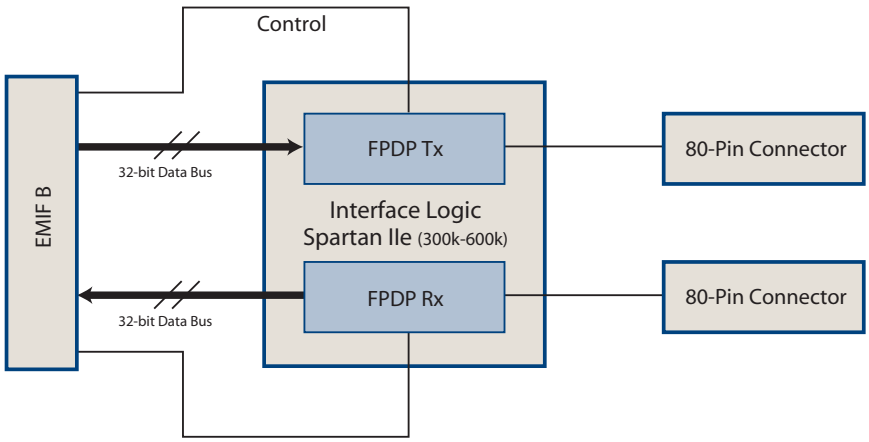


FIGURE 9. FPDP Overview.

The primary method used for moving data to the DSP and from the DSP memory is by using a DMA channel. In most cases, DMA delivers data in the most efficient method because it preserves DSP CPU bandwidth, is more efficient at bus utilization, and has a lowest interrupt latency. Using CPU accesses or DMA, single data point access may also be performed. Since the FPDP FIFOs are burst memory devices, the maximum write and read rate will be about 1/3 the speed of larger burst packets due to the transfer setup cycles inherent on the DSP burst memory access protocol.

Data Link Layer Specification

Many applications of FPDP are high performance data acquisition and transfer systems. These frequently deal with multiple channels of data. It is essential to have a method for identifying the channel associated with each item of data.The method chosen is to allow for data frames, where each frame is delineated by an assertion of the *SYNC(active low)* signal.

The data frame types defined by this standard are as follows:

- Unframed Data
- Single Frame Data
- Fixed Size Repeating Frame Data
- Dynamic Size Repeating Frame Data *(NOT Supported by the Software, yet)*

Unframed

Unframed Data interfaces are intended for use where the organization of the data is of no relevance and the FPDP/RM (i.e. FPDP Receiver Master) or FPDP/R (i.e FPDP Receiver) interface does not need to be informed of a synchronization point in the data stream. When using unframed data, the *sync pulse* signal is *not* used.

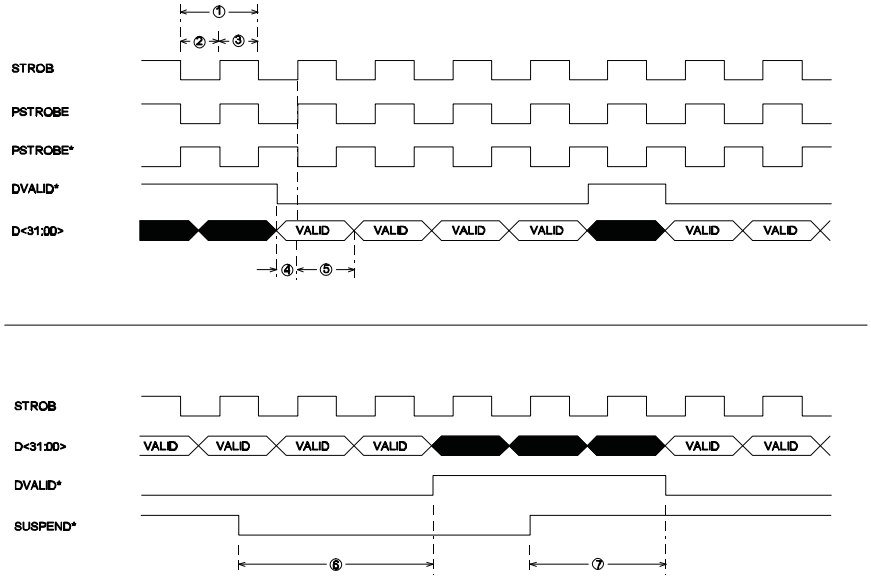


FIGURE 10. FPDP Timing Diagrams for ALL Data Framing Types.

Single Frame

The single Frame Data and the two Repeating Frame Data type definitions serve the requirements of interfaces, which must pass data in response to an event, such as multi-channel A/D converters, devices that transfer fixed length data files. This is accomplished by synchronizing data acquisition at FPDP/RM or FPDP/R interface to a separate signal, *sync pulse*.

In Single Data Frame Data, the synchronization is intended to occur between blocks. As a result, Single Frame Data compliant interfaces require that the synchronization event occurs prior to the data being synchronized, before the assertion of DVALID *(i.e. Data Valid - active low).*

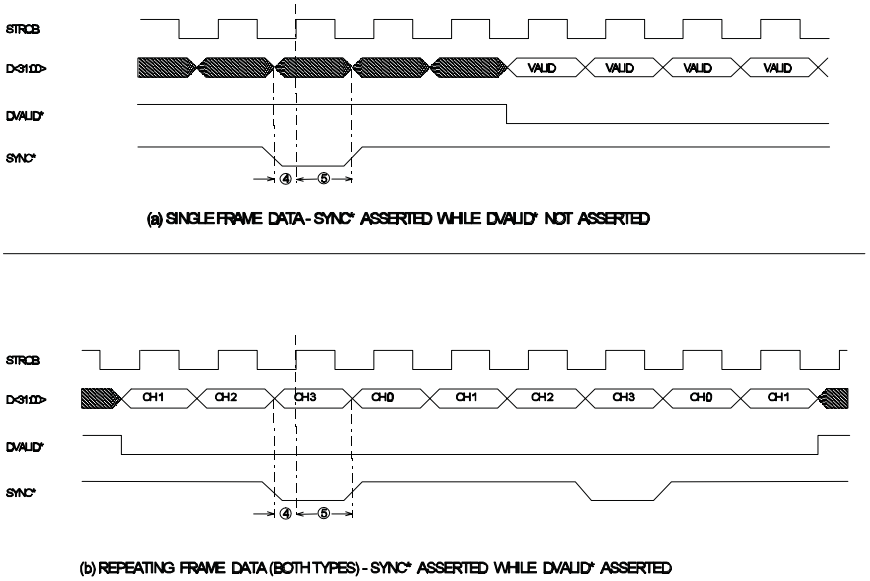


FIGURE 11. Standard FPDP Timing Diagram For Single Frame And Repeated Frame Data.

Fixed And Dynamic Size Repeating Frame

Both of the Repeating Frame Data Types, the synchronization event occurs coincident with the last data word transferred in the block before. This means that reception of the first frame may not be synchronized and it should be ignored by the receiving board. Both cases requires the synchronization event occurs at the end of the block prior to that being synchronized, while DVALID *(i.e. Data Valid -active low)* is still asserted.

The difference between both repeating frames is that fixed size repeating frame permits only frames of the same length, whereas the dynamic size repeating frame sizes to vary from frame to frame in an arbitrary manner.

CE Space	Address	Function	R/W	Type
1	0x90002200	FPDP Tx PIO	R/W	Async
1	0x90002300	FPDP Rx PIO	R/W	Async
1	0x90002100	FPDP Rx Frame Count	R	Async
1	0x90000C00	FPDP Rx Config	W	Async
1	0x90000D00	FPDP Tx Config	W	Async
1	0x90000E00	FPDP Tx Frame Count	W	Async
2	0xA0000000	FPDP FIFO	R/W	Burst

TABLE 17. FPDP Memory Map

Bit	Function
0	FPDP FIFO Reset '0' Reset '1' Out of Reset
2-1	Sync Mode "00" Unframed
5-4	PIO Direction PIO signals are input until configured as outputs. "11" For Outputs.
16-8	Threshold

TABLE 18. FPDP Rx Configuration Register.

Bit	Function
0	FPDP FIFO Reset '0' Reset '1' Out of Reset
2-1	Sync Mode "00" Unframed "01" Single Frame "10" Fixed Repeating "11" Dynamic Repeating
5-4	PIO Direction PIO signals are input until configured as outputs. "11" For Outputs.
16-8	Threshold

TABLE 19. FPDP Tx Configuration Register.

'C6713 McBSP Serial Ports

The 'C6713's on-chip serial ports (McBSP) are pinned out to connectors JP14 (port 0) and JP15 (port 1) for use with external hardware. The serial ports are also connected to the OMNIBUS sites for use with modules. Omnibus module 0 has McBSP port 0 and module 1 has McBSP port 1.

Pinouts for the serial port connectors are given in the appendices. Innovative Intergration recommends buffering these ports with off board hardware in order to preserve signal integrity.

PCI Interface

The M6713 has a flexible PCI bus interface that supports control and configuration by the host computer and data transfers at up to 512 MB/s. The DSP and host can efficiently transfer data to one another using DMA operations over the PCI bus as a bus master. The DMA controller uses a data packet protocol and credit-based flow management to control the data movement and provide an efficient interface. The controller and its software layers simultaneously support both high speed sustained data transfers and on-demand data transfers through use of the unique credit management system and interrupts.

As shown in the following diagram, the DSP communicates with the PCI DMA controller through a pair of 2kB FIFOs. These FIFOs are mapped into the DSP memory space as burst memories and support transfer rates of up to 300 MB/s to the DSP. The DSP transfers data based upon a DMA interrupt driven by the availability of data for each packet transferred.

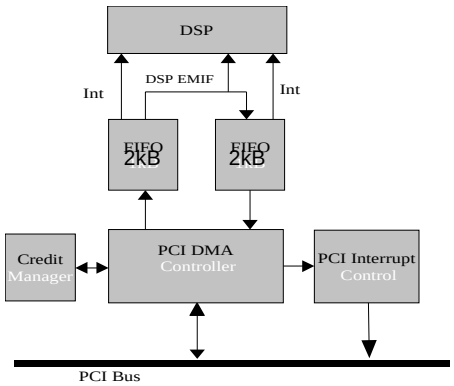


FIGURE 12. PCI Interface Block Diagram

PCI DMA Data Rates

The DMA transfer rates vary, according to the bus speed, buffering, bus efficiency and other factors. For our software system with the M6713, here is a guideline of what can be expected for systems we tested.

PCI Bus Clock	Data Width	Theoretical Rate (MB/s)	Typical Sustained DMA Rate (MB/s)	Comments
33.3	32	133	80	This is a typical desktop PC. Rates vary from 50 to 120 depending on the bus efficiency and system speed.
33.3	64	266	120	Workstation and servers are typical platforms for this bus. Rates vary from 90 to 200MB/s for machines measured.
66.6	32	266	120	Don't see many of these. Measured 120 MB/s on the system we tested.
66.6	64	512	280	Servers and embedded systems typically have this bus type. Rates vary from 200 MB/s to 320 MB/s.

TABLE 20. PCI DMA Rates Summary

As would be expected, the better architectures and performance of servers result in the better performance. In some cases, the software can be a limiting factor, especially as packets get small and interrupt rates increase. To get best performance, bigger packets and better machines are always better.

Compatibility. The bus interface is compatible with PICMG 2.1 standard, supporting 3.3V or 5V signaling, up to 66 MHz clock rates and 32 or 64 bit operation. The bus interface logic automatically detects the type of bus and configures itself accordingly. PCI-X slots are OK to use also since they will work as PCI as required by the specification.

Packet Transfer Protocol. The PCI DMA interface uses data packets for transporting the data. Each packet has a header and a data payload that carries the data. Data sent to the PCI DMA controller MUST be in this format for proper operation. Each packet is inspected by the PCI DMA controller as part of the DMA process. The controller moves the number of points in each packet by counting the number of points transferred. An interrupt is given o the host for each packet, as determined by the header.

Packet Format. The packet header consists of a 24-bit field for the packet size and a Peripheral Device Number. The packet size is given in 32-bit words and includes the 2 word header. The Peripheral Device Number may be used by the DSP or host to implement further protocol, such as channelization or message passing, but is simply ignored by the PCI controller.

FIGURE 13. PCI Packet Format

Word	Description

1	Bits 31..24 Peripheral Device Number - used for backend processing Bits 23..0 Packet size in 32-bits including header
2	Not used
3..N	Data Word

Host DMA Software Support. Software methods are provided to control the PCI busmastering process. The software implements all the controls necessary to efficiently use the PCI bus mastering capabilities of the M6713. Interrupts are serviced in the device driver, providing maximal efficiency for the host. Credit and data management by the software allow the applicaiton to interact with the M6713 by managing credit for the data flow.

The M6713 Development Package contains extensive driver support for the PCI bus environment. Users are strongly encouraged to make use of the standard software drivers in order to speed product development.

If you wish to target an operating system not supported by Innovative, we encourage you to discuss this requirement with our technical staff. Many details involved in the use of the PCI busmastering interface are not obvious.

DSP Software Support for Using the PCI DMA Controller . The DSP can use the PCI bus interface DMA controller by writing to the FIFO, or reading from the FIFO when an interrupt signals that data is available. The data packets must be formatted with the 2 word header as described.

Interrupts to the DSP are used in both data flow directions to pace the data movement. In the DSP to host direction, the DSP sets a FIFO level at which an interrupt will occur in the PCI FIFO level control register (CE0, 0x80180000) , signaling that there is more room in the FIFO for data. A DMA channel on the DSP is used to move data based on this interrupt to the FIFO (CE1, 0x90000000). In the host to DSP direction, the DSP moves data from the FIFO (CE1, 0x90000000) when an interrupt signals that a packet is available. The DSP then reads the header and programs a DMA channel according to the packet size to move the data from the FIFO to the DSP.

The FIFO management requires that the DSP use the PCI FIFO level control register to control the interrupts from the FIFOs. The FIFO is 256 deep, so large packets must be split into pieces, usually 32 to 128 words in size for the DSP DMA channel to move. The DSP DMA channel is programmed to move the entire packet size as small pieces, driven by an interrupt from the PCI FIFO level, with a final terminal count interrupt when the packet move is completed.

Timers

The M6713 provides a total of two counter/timers as well as the DDS used for timebase generation. These timers are independent and may be used as on board timebase generation for use in timing data acquisition, servo controls, real-time counters, and many other applications. The counter/timer func-

tionality are two 32-bit timer channels on the ‘C6713 processor and a 32-bit direct digital synthesizer (DDS) channel in the AD9851 .This section discusses the AD9851 synthesizer in detail: for more information on the on-chip timers, see the *TMS320C6000 Peripherals Reference Guide (SPRU 180)*.

On-chip Timers

The on-chip DSP timers are available for use as software timebases and interrupt generators. Note that the ‘C6713 does not support timer operation during JTAG emulation halt when the timers are programmed to use an external source. When clocking from an external signal, halting the DSP from a JTAG debugger will also halt the on-chip timers. This does not apply when the timers are run from the processor clock (timers will run when the CPU is halted).

AD9851 Direct Digital Synthesizer

The AD9851 direct digital synthesizer (DDS) is a precision programmable clock source which is capable of generating frequencies in the range of 0 to 25 MHz with a resolution of 0.019 Hz/step. Unlike a digital counter-timer chip, which uses a digital counter to divide down a high input clock rate, the DDS uses phase-locked-loop synthesizer technology to tune a sine wave oscillator based on a 32-bit digital word. This method realizes a linear output frequency over input range rather than the nonlinear one associated with counter-timer chips, whose resolution drops dramatically as the period register used to program them falls. The DDS should be used when a precise and accurate clock is required by the application.

DDS Jitter Reduction. The DDS has the lowest jitter when operating close to its highest frequency of 25 MHz. Jitter is typically 100 pS at 20 MHz, but is substantially worse at low frequencies. This jitter is usually only of concern for high frequency analog sampling (>10MHz) or when the clock is used as a PLL reference. The DSP timer clock outputs are lower jitter typically.

To reduce the jitter, the M6713 has a post-scaler for the DDS in the logic. The post-scaler divides the DDS output by 1 to 65536.

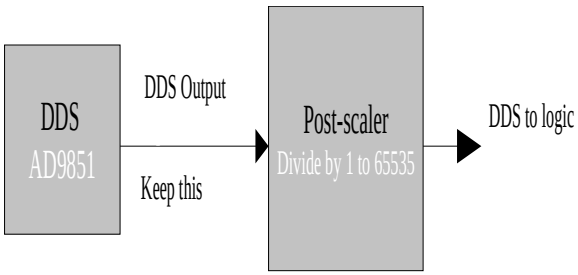


FIGURE 14. DDS and Post-scaler

The software support package for the M6713 adjusts the DDS frequency to be as close to 25 MHz as possible, then divides the output by a post-scale value. No resolution restriction is created by this post-

scaling since the DDS has very fine resolution and when used with the post-scaling integer division still results in the full adjustment range. The software computes the optimum value for to get low jitter from the DDS.

Bits	Function
15..0	DDS post-scaling value. The DDS is divided by this number. DO NOT enter a 0. Default is 0xFFFF; 0x1 = no postscale, no divide
31..16	Not used

TABLE 21. DDS Post-dscaling Control Register (Module 0 0x8020002C, Module 1 0x80200030)

Note that there is a post-scaling register for each Omnibus module site allowing different post-scaling for each module.

DDS Control. The AD9851 is mapped into memory as shown in the table below. The device is interfaced using the parallel I/O method, with one address to write data, one to trigger frequency/phase updates, and one to control the reset pin of the device.

Function	I/O Space Address
DDS Control	0x80020000
DDS Data	0x80030000

TABLE 22. AD9851 Control Registers

The write clock address latches frequency/phase data into the AD9851 one byte at a time. The least significant eight bits of the processor bus carry the byte wide data. The frequency update address causes the output frequency and phase of the DDS clock to update to the values contained in its input latches. The reset address causes an active high reset pulse to be generated to the AD9851. By default, the DDS reset is true at power-on or reset, so a ‘0’ must be written to the DDS reset control bit (bit 0 on DDS Control register) in the control register before configuration and use. These registers are write-only.

The output of the DDS may be used for a variety of functions including driving interrupts, as a timebase to the DSP or modules, or as an output. Refer to the register descriptions for interrupt use, or timebase pin definition registers.

The M6713 Development Package includes support routines, which make it easy to set the AD9851’s output frequency as discussed in the previous sections of this manual.

The DDS has very fine resolution allowing the application to tune the timebase to many frequencies. The absolute accuracy of the DDS timebase is approximately 200 ppm for room temperature applications. This absolute accuracy may vary with temperature and time. Calibration may be required in applications requiring higher precision timebases. Timer Output Control

The M6713 has a variety of timers, timebase controls. The timer0, timer1 and DDS pins therefore have software programmable outputs so that any of the available triggers can drive the modules. Historically, Innovative has referred to these pins by the names timer0, timer 1, and DDS which reflected their dedi-

cated function, but now the “timer0” pin can have any number of signals on it, not just the traditional timer0 function.

That being said, the register to program what signal is on the timer 0 and 1 pins delivered to the Omnibus with the following bit assignments.

Omnibus Pin	Bit	Function
timer0	0	DSP timer0 Output
timer0	1	DSP timer1 Ouput
timer0	2	SyncLink(0)
timer0	3	SyncLink(1)
timer0	4	ClockLink In
timer0	5	External Clk 0
timer0	6	External Clk 1
dds0	16	DSP timer0 Output
dds0	17	DSP timer1 Ouput
dds0	18	External Clk 0
dds0	19	External Clk 1
dds0	20	SyncLink(0)
dds0	21	SyncLink(1)
dds0	22	ClockLink In
dds0	23	DDS chip output

TABLE 23. Timer0 and DDS Pin Output Sources (Module 0 0x80200034, Module 1 0x80200038)

Digital I/O

The M6713 includes 32-bits of software programmable digital I/O for use in controlling digital instruments or acquiring digital inputs. If you need additional bit IO, the FPDP ports may be used as bit IO also.

The Digital IO Port is a set of simple input and output latches. Outputs are enabled on a byte basis as programmed by the DIO control register. The input latch may be enabled to capture the pins whenever it is not being read, or whenever an external signal (EXT DIG CLK) is low, as enabled by the Digital IO control register. When used in the internal enabled mode, the value read is the state of the pins immediately before the DSP reads them. The external clock signal is actually an enable to the DSP EMIF clock use to clock the latch, so it must be high a minimum of 27 nS to guarantee its operation.

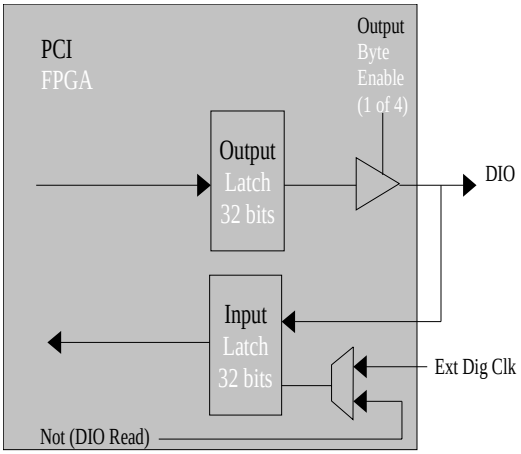


FIGURE 15. Digital IO Port Block Diagram

The digital I/O port controls are mapped into memory space using two addresses: one to read/write the digital I/O data as a single 32-bit word, one for direction control and for each byte of the port and controlling the source of the clock edge used to latch input data into the digital I/O port register. The following table lists the addresses and their functions.

Function	Address
Digital I/O Data Register	0x80010000
Digital I/O Direction Control and input latch clock control	0x80000000

TABLE 24. Digital I/O Control Registers

The direction control register provides for software control of the drive direction of the port and the source of the input latch clock. The least significant four bits of the register control the four bytes available on the I/O port. Bit D0 sets the direction for the least significant eight bits if the port (port bits 0-7), D1 the next least significant bits (8-15), D2 the next least significant (16-23) and D3 the most significant (24-31). Each byte is individually controllable by writing a zero (to select output) or a one (to select input) to the respective bit in the direction control register. For example, if the value 0xC were written to the direction control register, bits 0-15 would act as inputs while bits 16-31 would act as outputs. All bytes default to input mode upon the board power-up or on reset.

Bit	Function
0	Direction Byte 0 0= Output (default), 1 = Input
1	Direction Byte 1 0= Output (default), 1 = Input

Bit	Function
2	Direction Byte 2 0= Output (default), 1 = Input
3	Direction Byte 3 0= Output (default), 1 = Input
4	Input Latch Clock 0 = Latched on Digital Read 1 = Latched on External Digital Read Clock

TABLE 25. Digital IO Port Control Register Bit Definitions

The data register allows software to directly read data from port pins programmed for input, or write data to pins programmed for output. Read operations performed from the data register on port bytes programmed for output will return the current value of the digital I/O latch (i.e. the last value written to that portion of the port). For example, suppose that the direction controls were programmed to 0xC and the data register written with the data word 0x12340000. Since the most significant 16-bits are setup as outputs, those pins on the port connector would assume the value 0x1234. A subsequent read of the port would yield the value 0x1234xxxx, where xxxx is the value of the signals present on the digital I/O connector.

The input latch clock bit allows the user to select from either software read clocking or external hardware clocking. Writing a zero to the register selects software clocking, while writing a one selects external hardware clocking. If software clocking is selected, then the port latches programmed for input will clock in the digital data present on the external pins at the beginning of a read cycle executed on the port data register (30-50 ns before the data is returned to the processor, depending on processor clock speed). If external clocking is selected, then the port will latch data on the falling edge of the TTL signal EXT_DIG_RD_CLK* at the digital I/O connector. The data will be held for the processor to read until the next low-going edge of the EXT_DIG_RD_CLK* signal. In the external hardware clocking mode, read operations by the processor do not affect the contents of the digital I/O latch. The latched data may be reread as many times as is required, and only another EXT_DIG_RD_CLK* pulse will cause new data to be latched into the port.

Digital I/O Timing

The following diagram gives timing information for the digital I/O port when used in external readback clock mode (see above for details). This data is derived from device specifications and is not factory tested.

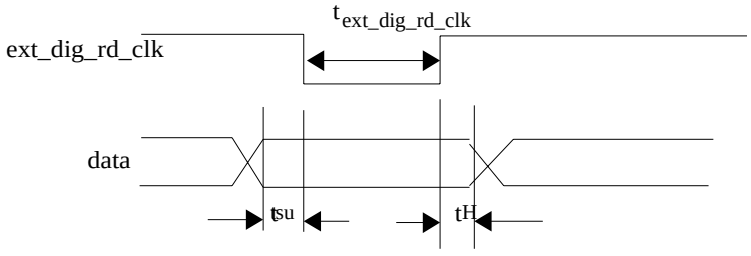


FIGURE 16. Digital I/O Port Timing

Parameter	min (ns)
t_{SU}	0
t_H	10
$t_{ext\_dig\ rd\_clk}$	27

TABLE 26. Digital I/O Port Timing Parameters

Digital IO Electrical Characteristics

The digital IO pins are TTL compatible pins driven by LVTTL (3.3V) pins on the FPGA. Each pin has a 100 ohm series resistor to allow 5V tolerant IO, per Xilinx requirement for the Spartan3 FPGA. The pins will drive 3.3mA per pin when this resistor is used. High current devices may need to provide additional current driving capacity.

Interrupts

The ‘C6713 processor implements five interrupt input pins, plus one GPIO pin configured as a DMA interrupt, that allow external hardware events to directly trigger software or DMA activity. Processor interrupt inputs are supported on the M6713 through a set of control registers and multiplexers that allows application software to dynamically select the source of the signal which will drive each particular interrupt input as well as the polarity and edge or level sensitivity. Additionally, the interrupt controller in the support logic allows interrupt sharing by peripherals on the M6713.

The following table shows the addresses of the control registers for each processor interrupt input. A value written to the appropriate control register causes the interrupt mux to select the interrupt source given in the next table (see below).

Functional	Address
NMI Interrupt Input Select	0x800D0000
External Interrupt Input 4 Select	0x80090000
External Interrupt Input 5 Select	0x800A0000
External Interrupt Input 6 Select	0x800B0000
External Interrupt Input 7 Select	0x800C0000
DMA Interrupt Select	0x80070000

TABLE 27. External Interrupt Input Control Register Addresses

Bit	Interrupt	Description
0	PCI write FIFO int	Interrupt to DSP when PCI write FIFO is below the set threshold
1	PCI read FIFO int	Interrupt to DSP when PCI write FIFO is above the set threshold
2..13	Not used	-
14	BM rd packet end (all data rcv'd in the FIFO on a read packet)	An interrupt to the DSP indicating that the busmaster controller has retrieved all the data in a packet
15	Not used	-
16	Mod 0 Int 0	Omnibus Module 0, int 0
17	Mod 0 int 1	Omnibus Module 0, int 1
18	Mod 1 Int 0	Omnibus Module 1, int 0
19	Mod 1 Int 1	Omnibus Module 1, int 1
20	Ext Clk 0	Interrupt from the Ext Clk SMB input 0
21	Ext Clk 1	Interrupt from the Ext Clk SMB input 1
22	fpdp_tx_fifo_int;	FPDP TX port FIFO is above the set threshold
23	fpdp_rx_fifo_int;	FPDP RX port FIFO is below the set threshold
24	fpdp_rx_sync_int;	FPDP Rx port interrupt when a sync is received- the number of points received in the frame may be read
25	Sync 4	SyncLink input 4
26	Sync 5	SyncLink input 5
27	fpdp_tx_next_frame_int	FPDP Tx port is ready for the next frame size
29..27	Not Used	-
30	Acknowledged mode (NMI and INT4 only) 0= non-ack'd(default), 1 = ack'd	Sets acknowledged mode if true which requires that each interrupt source be acknowledged with a write to the interrupt ack register.
31	Enable NMI Interrupt (NMI only)	'0" = disabled (default)

TABLE 28. External Interrupt Control Register Bit Definitions

The selection bits in each interrupt control register are identical on all interrupts. Set the corresponding bit true for each interrupt source that is used by that DSP interrupt. For example, if module 0 interrupt 0 is the interrupt source for the DSP external interrupt 4, then a 0x1000 must be written to the DSP inter-

rupt 4 select register at 0x80090000. Multiple interrupts sources may be enabled on shared interrupts. Dedicated interrupts should enable only one interrupt source. At reset, no interrupt sources are enabled.

Bit 30 is the mode selection for acknowledged interrupts mode described in the section on shared interrupts.

Conditioning for Interrupt Input Signals

Each interrupt source has polarity and edge/level selection so that nearly any interrupt source can be used by the DSP interrupts. The interrupt polarity is controlled by the interrupt source polarity register at 0x80110000 with each interrupt source, as numbered in the interrupt selection register above, controlled by a bit in the register. Edge/level selection allows the DSP to use either the interrupt source edge or level as the trigger condition for the interrupt. Polarity selection is normally used to control the edge used, rising or falling, or the level, high or low, for an interrupt. The following table gives the correct settings for the possible interrupt conditions so that the logic interacts properly with the DSP. The M6713 support logic always gives a rising edge interrupt to the ‘C6713 DSP, thus the DSP requires a rising edge interrupt configuration.

Interrupt Type	Polarity	Edge/Level
Rising Edge	1	0
Falling Edge	0	0
High Level (1 = interrupt)	1	1
Low Level (0 = interrupt)	0	1

Interrupts must remain active at least 56 ns after changing states, in either edge or level mode.

Shared/Dedicated Interrupts

The M6713 has two interrupt modes (only on on the NMI and INT4) that may be used with any processor interrupt : shared and dedicated mode. Since the DSP has only five interrupts, many applications need a method for sharing interrupts to support all the peripheral devices on the M6713 efficiently. Shared mode has been developed so that multiple interrupt sources may share a single interrupt to the DSP.

In dedicated mode, each interrupt to the processor is steered directly from the selection matrix, through the edge/level and polarity conditioning directly to the processor. This allows the interrupt source to directly connect to the DSP. Dedicated interrupts are not shared amongst interrupt sources and should be used for the devices requiring the highest rates of interrupt servicing. These devices might be, for example, FIFO level interrupts from an Omnibus module or FPDP interrupts. The dedicated mode does not have the burden of acknowledging the interrupts consumed so this mode is faster at interrupt servicing than the shared interrupts.

Shared interrupts allow multiple devices to share an interrupt to the processor. When the DSP receives a shared interrupt, it must read the interrupt status register associated with that interrupt to determine the

interrupt source(s) requiring service. The DSP interrupt handling in this case should be capable of handling all the devices sharing this interrupt either alone or simultaneously to support the interrupt sharing. Upon completing the interrupt servicing, it is required that the DSP acknowledge the interrupt sources that were serviced. This prevents interrupts from being lost in the event that another interrupt source requires service in the meantime. The interrupt status/acknowledge registers are located at the address in the following table. Writing a '1' to any of the bits indicates that the interrupt has been serviced.

The bits have the same order as the interrupt source selection bits previously shown above.

Interrupt	Address
DSP int 4	0x80050000
NMI	0x80080000

TABLE 29. External Interrupt Status and Acknowledge Register Addresses

For example, if the application requires the output from external interrupt input two to drive processor interrupt input four, the value two should be written to memory location 0x80050000. All interrupt control registers default to 0 on power-up or board reset. Note that the processor interrupt signals generated by the logic are active high (rising edge trigger), and the 'C6713 interrupt polarity control register must be programmed to the value 0x0 to correctly receive interrupts.

Using Interrupts for DMA

To use the interrupts to drive a DMA channel on the DSP, it is necessary to control the behavior of the interrupt to prevent false interrupt triggering. This often occurs when an interrupt is used to read data from a FIFO. The typical arrangement is to set a level threshold for the FIFO and to trigger a DMA interrupt when the threshold is crossed. The only problem is that the FIFO may also be simultaneously being accessed from the other side, so the level can fluctuate as the read/write accesses occur. This can cause extra interrupts.

To control this behavior, an access counting register is added so that once the interrupt signals, a programmed number of accesses must occur before another interrupt is issued. Our internal parlance for this the interrupt "burst counting" register. This is because the DMA typically reads a burst of data from the FIFO or other device and we count the accesses in the burst. The DMA controller in the DSP is programmed for a transfer count that is also used to program the interrupt burst counting register. When the DMA interrupt signals, the DMA controller will read the data from that address for the programmed count

The M6713 logic has generalized this function so that a read or write to a particular is counted for the interrupt contorl. This allows any peripheral device to take advantage of the interrupt counting for DMA control. Each interrupt has a register set for the address, and a control register for the access direction and enable.

Bit	Function
6..0	Burst count

29..7	Not used
30	DSP read or write access 1 = read; 0 = write (default)
31	Enable DMA burst counting interrupt mode

TABLE 30. Interrupt Burst Counting Control Register

Bit	Function
31..0	DMA address for burst counting

TABLE 31. DMA Address for Interrupt Control

Address	Register Description
0x801B0000	Burst address register – INT4
0x801C0000	Burst address register – INT5
0x801D0000	Burst address register – INT6
0x801E0000	Burst address register – INT7
0x801F0000	Burst address register – DMA int
0x80120000	Burst count control register – INT4
0x80130000	Burst count control register – INT5
0x80140000	Burst count control register – INT6
0x80150000	Burst count control register – INT7
0x80160000	Burst count control register – DMA int

TABLE 32. Interrupt Access Counting Control Registers

For the drivers provided in the M6713 software development package, this interrupt control is implemented as part of the peripheral driver under DSP BIOS.

External Inputs for Clocks and Interrupts

The M6713 supports two external inputs via SMB connectors J1 and J2 that may be used as input tirggers, clocks or interrupts. These inputs are 50 ohm terminated as selected by jumpers (see table)2 (jumper on = 50 ohm input) or unterminated (100 ohms in series to the FPGA pins). These pins are a

direct connection to the FPGA and is expected to be LVTTL compatible. The inputs are 3.3V tolerant. Overvoltage protection for ESD and transient protect limits the inputs to 0 to 3.3V.

Jumper	External Input	Impedance
JP2	Ext Clk 0	No jumper = Hi Z Jumper = 50 ohms
JP11	Ext Clk 1	No jumper = Hi Z Jumper = 50 ohms

TABLE 33. External Clock Input Termination Jumper Settings

Within the standard logic, either of these inputs may be used as a trigger or clock source and may be shared to other cards over the synclink/clocklink connections. Custom logic designs may use the connections as either inputs or outputs.

Multi-Card Timing Synchronization

The M6713 has several features to support multi-card synchronization and clock sharing. Synclink is a simple TTL bus that allows the M6713 to send or receive clock and trigger signals to other cards, while ClockLink is an LVDS input and output pair allowing the system to share high speed clock and trigger signals. A variety of DSP and data acquisition products from Innovative feature SyncLink and ClockLink to allow the construction of multi-card synchronized systems for channel expansion and common triggering.

The SyncLink bus has one master who is the originator of all clock and trigger signals, referred to as the master, while all other cards are referred to as slave devices. The slave cards can only receive clock/trigger signals from the master. The following table below shows the signals that may be transmitted over the SyncLink bus when it is the master. The SyncLink signal selection register resides at 0x80200004.

Bit	Signal Output Control
0	Sync Master

Bit	Signal Output as SyncLink 0
1	DSP timer 0
2	DSP timer 1
3	Module 0 DDS Clock
4	External clock 0 (J1)
5	not used

TABLE 34. SyncLink 0 Signal Selection Register Bit Definitions

Bit	Signal Output as SyncLink 1
6	DSP timer 0
7	DSP timer 1
8	Module 1 DDS Clock
9	External clock 1 (J2)
10	not used

TABLE 35. SyncLink 1 Signal Selection Register Bit definitions

Bit	Signal Output as SyncLink 2
11	Module 0 Trigger
12..15	not used
16	Module 1 Trigger
17..20	not used
21	Module 2 Trigger
22..24	not used
25	Module 3 Trigger

TABLE 36. SyncLink 2 Signal Selection Register Bit definitions

The SyncLink master bit is controlled by the DSP at address 0x80200004. Write a '1' to bit 0 of this register to allow the M6713 to act as the SyncLink master, thus making it the source of the synclink signals. Only one master should be enabled between all the cards on the SyncLink bus to prevent conflicts.

The SyncLink signals are recommend for use below 2 MHz to a maximum of 8 TTL loads. Termination may be required depending on the cable and impedance of the loads to prevent signal ringing. More than eight loads may require additional buffering.

ClockLink is intended for higher speed signals that are usually card to card. The ClockLink allows the M6713 to share clocks up to 80 MHz over short distances, or farther for slower signals. Flat ribbon cable performs well, but twisted pair cable is recommended for the best signal integrity. Note that the transmit/receive signal wire pairs must be crossed to connect two cards together.

ClockLink may source a variety of signals as shown in the following table controlled by the Clocklink Configuration Register at 0x8014000.

Bit	Signal Output as SyncLink 0
0	DSP timer 0
1	DSP timer 1
2	External Interrupt Signal (J2)

Bit	Signal Output as SyncLink 0
3	Timer Clock (80 MHz)
4	DDS Output

TABLE 37. ClockLink Signal Selection Register Bit Definitions

JTAG Test Bus

The M6713 implements a JTAG 1149.1-compatible scan path loop through the onboard ‘C6713, with connector compatible with the specification provided in the *TMS320C6000 User’s Guide*. JP16 is the JTAG connector for use with a JTAG controller card cable (from an Innovative Integration Code Hammer debugger card, Texas Instruments XDS-510, or other vendor’s JTAG hardware).

Power Requirements

The M6713 uses the PCI bus power for operation. On-card power supplies make some of the unique voltages required by the M6713 for the DSP, FPGA and analog IO from the host power supplied over the PCI bus.

Supply Voltage	Current	Use
+5VDC	3A max; 0.82A typical	All digital electronics including DSP and FPGA
+12V	Module dependent	Omnibus Supply - usually for analog IO
-12V	Module dependent	Omnibus Supply - usually for analog IO

Updating the M6713 logic

The M6713 logic may be updated in the field under normal circumstances without removing the card from the system. There are two FPGA logic devices on the card : the PCI control logic and the Interface logic. The Interface logic must be loaded each time the M6713 is powered-up since there is no on-card memory is provided for the Interface logic FPGA. The PCI controller logic has an on-card FLASH memory that holds its logic image and may be reprogrammed in the field.

Reprogramming the PCI Controller Logic. The PCI FPGA logic is held in a reprogrammable FLASH that can be written by the DSP so long as the DSP card is functioning normally prior to re-programming. In this case, an application provided with the Pismo Toolset allows the user to load new logic to the card over the PCI bus link.

A few precautions should be taken so that the card is not left in an unrecoverable state from an unsuccessful programming attempt. First, the power should remain on during the entire logic burning process as partial rewrites will cause failures. Second, the card must communicate normally prior to burning the logic or a faulty logic image may be downloaded. Third, use only logic image files (.BIT) provided by Innovative for the M6713 during the update process.

Warning: Should the card ever fail during the update, or if it is not functioning normally, contact technical support at Innovative. Do not attempt logic updates under this condition or the card could be rendered useless.

Updating the logic is only supported using the logic update program from Innovative included in the Pismo toolset.

Reprogramming the Interface Logic. The Interface logic may be reprogrammed at any time on the M6713. No on-card logic image is stored for the Interface logic and it must be loaded after each power-up.

The Interface logic is loaded over the SelectMap Interface from the PCI bus or over JTAG. The JTAG download is usually used during the logic development and debug process. The SelectMap Interface is mapped to the PCI bus. The M6713 Interface logic can be downloaded using either the M6713 downloader program using the M6713_intf.EXO file PROM image.

If you are developing custom logic for the M6713, see the FrameWork Logic User Guide that gives detailed instructions on downloading logic images over JTAG or using the downloader program.

Making Custom Logic

See the FrameWork Logic users guide for developing custom logic and MATLAB use.

CHAPTER 10

Troubleshooting

Initialization Problems

Borland C++Builder Problems

The Builder Help Index system is blank.

Borland uses the standard Microsoft help system for its online help. Normally this works perfectly, however, Windows 9x Help system has a limitation on the amount of references it can use simultaneously. If you have too many links in your help index, you will not see any help at all on the index. However, if you are in Builder and you use the F1 functionality for your help, it will work perfectly.

To get your Help index working again, you will need to slim down the contents of the Open Help Index list found. Open Builder.

- Bring up Open Help by selecting *Help* | *Customize*.
- Click the **Index** tab. Remove files until your Index Help is working.
- Remember to Save the project before checking the Index Help.

To add Innovative help files if they did not automatically get added, follow the instructions above, except add help files. All help files can be found at *Program Files* | *Innovative* | *Documents* | *manuals*. The files to be added should **not** include files with 'Pismo' or 'Zuma' in their names as these are target side help files.

I created an EXE file and when I try to run it, the system requires a DLL which I don't have.

Depending on the settings your application has for building, it may require certain DLLs, such as **borlndmm.dll**. When you try to run your newly created executable,

you may get an error such as “Dynamic link library borlndmm.dll can’t be found”. One cause of this is when you have the project set for dynamic rather than static linking of the Borland VCL packages. While dynamic linking can result in smaller EXEs, dynamic linking can result in dynamic link error messages, such as the one mentioned above, when a DLL is unavailable or not find-able by the application program at invokation.

We recommend static-binding of all executables. To do this:

- Bring up the Project Options menu by clicking the Project | Options menu in Builder. Select the **Linker** tab.
- Deselect the “Use dynamic RTL” option. This will force the inclusion of the DLLs that are required by your application.

Another possibility is that you have built your application with runtime packages. By building in this fashion, the executable requires certain BPLs to be in the system directory. To build your application without this dependency, click the **Packages** tab on your Project Options menu. Deselect “Build with runtime packages” option.

What DLLs do I have to deploy with my newly created executable?

The following DLLs must reside on the path for any deployed Vista application: Typically, these files are placed in the Windows system directory.

For Windows 9x, the system directory is the **C:\windows\system** directory. For Windows NT, it is **C:\Winnt\system32** directory.

Function	Required DLLs
Intel native signal processing libraries	nsp.dll, nspa6.dll, nspm5.dll, nspm6.dll, nspp6.dll, nsppx.dll, nspw7.dll
Intel native image processing libraries	All files in the Innovative\Lib\Ipp directory tree
Innovative device driver DLL	iidrvx40.dll
Code Hammer Debugger	IIPciPod.dll
Innovative Registration DLL	UserRegister6.dll (BCB users design-time only)

How do I know what DLLs my executable is dependent on?

The Applets\Third Party folder contains an archive call depends20_x86.zip which contains a utility called Depends.exe the provides a utility that allows you to determine the external dependencies of and Windows executable file. Clock File | Open from the menu bar and browse to the name of your application executable. The utility will list all DLLs on which your application is dependent in order to run. Note, however, that Windows programs are always dependent on Windows system DLLs, such as **Advapi32.dll, Kernel32.dll, Version.dll, Comctl32.dll, Gdi32.dll, User32.dll, Ole32.dll and Oleaut32.dll**. These dependencies cannot be eliminated, but will not cause a runtime error, since Windows systems allways provide these DLLs.

If the utility exposes DLL dependencies that you would like to remove, follow the steps in the question above “I created an EXE file and when I try to run it, the system requires a DLL which I don’t have.”

DSP Hardware Problems

The I/O seems like it is not connected or doesn't work.

Double-check the connections to the I/O connector. The most common error is to connect to the wrong pins on the I/O connector.

Check the trigger and timebase setups. For the analog I/O, be sure that the timebase you have selected is running, and the trigger method is valid. No data can be collected on these cards without a timebase and an active trigger (start trigger occurred, stop has not yet occurred).

How can I tell what version of logic I am using?

The logic versions are reported by the FlashBurn applet. When the applet is opened, the current bus and analog logic version numbers are displayed.

The I/O seems like it is not connected or doesn't work.

Double-check the connections to the module I/O connector. The most common error is to connect to the wrong pins on module connector. As described in the paragraph "OMNIBUS I/O Connector (JP4)", the pins on the module have a mapping to the end connector which means the pin numbers do NOT correspond one-to-one. The module pin numbers in the Omnibus hardware manual are the pin numbers on the mating connectors on the card itself. These internal pins are mapped to the MDR-100 type connector on the card end for user connections according to the Omnibus MDR-100 mapping table in the "Hardware" chapter.

How do I update the logic?

The logic may be updated using the FlashBurn program. This applet is capable of updating the PCI bus coprocessor Loader and Talker images, as well as the images for both the bus and analog FPGAs. The utility also supports embedding a C6713 executable, as generated by the PromImage.exe utility.

The updating process is straightforward and is generally trouble-free. Two big mistakes that can occur are burning the wrong logic into the Flash and terminating the application before completion. If you put the wrong logic into the card, this may be recoverable by just repeating the programming process. Provided that the logic file is from another version, this will usually work. If the file is completely wrong, DO NOT POWER CYCLE THE baseboard. So long as power is applied, the logic image in the FLASH is not yet used and you can still reprogram the correct one into the card.

If the program crashes or terminates early for any reason during the programming sequence, the card will almost certainly need to be returned to Innovative for reprogramming. Sorry.

I updated the logic, but it did not work.

The most common reasons for this are that the wrong logic image was used, or the card was not power-cycled between tests. The logic is not reloaded until the card is powered up again.

CHAPTER 11

Appendices

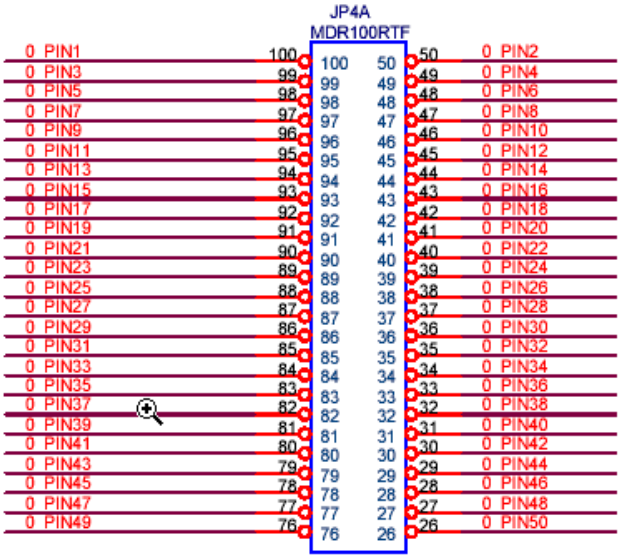
Connector pinouts

JP3, JP7 - OMNIBUS I/O Connectors

Connector types:	JP3, JP7: AMP .05 Subminiature D male, AMP 173280-3
	JP4 : MDR connector, 3M N102A0-52E2VC
Number of pins:	JP3, JP7: 50
	JP5: 100
Mating connector:	JP3, JP7: AMP 173279-3
	JP4: 3M 101A0-4CZ3JL

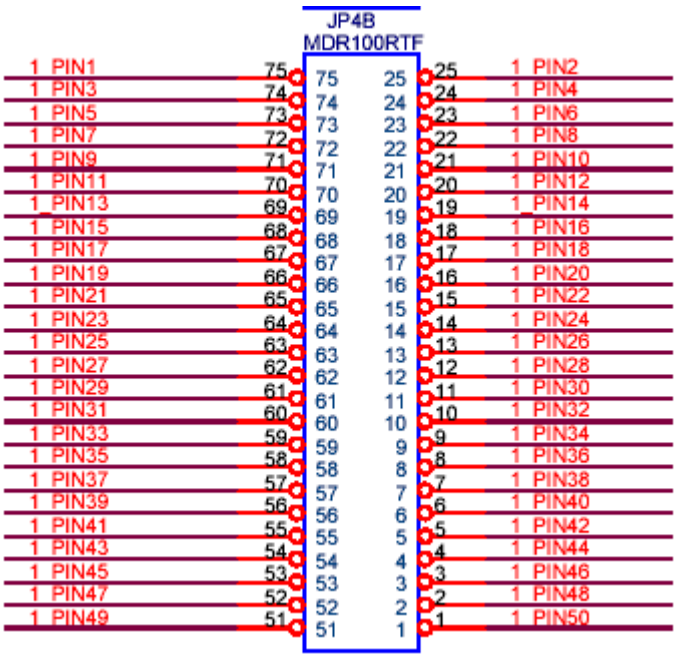
The following table shows the interconnections between the JP4 (OMNIBUS slot 0) and JP5 (OMNIBUS IO connector).

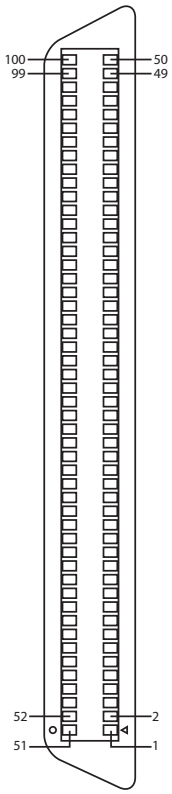
JP3, Module 0 Pin	JP4 Pin Numbers
1	100
2	50
3	99
4	49
5	98
6	48
7	97
8	47
9	96
10	46
11	95
12	45
13	94
14	44
15	93
16	43
17	92
18	42
19	91
20	41
21	90
22	40
23	89
24	39
25	88
26	38
27	87
28	37
29	86
30	36
31	85
32	35
33	84
34	34
35	83
36	33
37	82
38	32
39	81
40	31
41	80
42	30
43	79
44	29
45	78
46	28
47	77
48	27
49	76
50	26



The following table shows the interconnections between the JP8 (OMNIBUS slot1) and JP5 (OMNIBUS IO connector).

JP8, Module 1 Pin	JP5 Pin Numbers
1	75
2	25
3	74
4	24
5	73
6	23
7	72
8	22
9	71
10	21
11	70
12	20
13	69
14	19
15	68
16	18
17	67
18	17
19	66
20	16
21	65
22	15
23	64
24	14
25	63
26	13
27	62
28	12
29	61
30	11
31	60
32	10
33	59
34	9
35	58
36	8
37	57
38	7
39	56
40	6
41	55
42	5
43	54
44	4
45	53
46	3
47	52
48	2
49	51
50	1





JP5, 6, 9, 10 – OMNIBUS Bus Connectors

Connector types: AMP .05 Subminiature D male

Number of pins: 50

Mating connector: AMP 173279-3

The following table gives the pin numbers and functions for the JP6 (OMNIBUS slot 0) and JP10 (OMNIBUS slot 1) connectors. The functions for JP10 are identical to those of JP6, except where noted.

Pin Number	JP6 Function	JP10 Function	Direction (from M6713)
1, 19	Digital +5V		O, power
2, 20	Digital ground		O, power
3-18	Data bus 0-15		I/O
21, 43, 40, 45, 39, 26, 27	Address bus 2-8		O
28	Reset (active low)		O
29	External interrupt 0	External interrupt 2	I
30	Bus ready (active low)		I (open-collector)
31	Processor ECLK/2 (37.5 MHz)		O
32	DSP timer channel 0		O
33	R/W*		O
34	DDS timebase		O
35-38	IOMOD0-3 decoded selects (active low)	IOMOD4-7 decoded selects (active low)	O
25	Analog –15V (OMNIBUS –12V)		O, power
23	Analog +15V (OMNIBUS +12V)		O, power
41,42	Analog ground		O, power
22,24	Analog -15V		O, power
44,46	Analog +15V		O, power
47,49	Analog +5V		O, power
48,50	Analog -5V		O, power

TABLE 38. OMNIBUS Bus Connectors

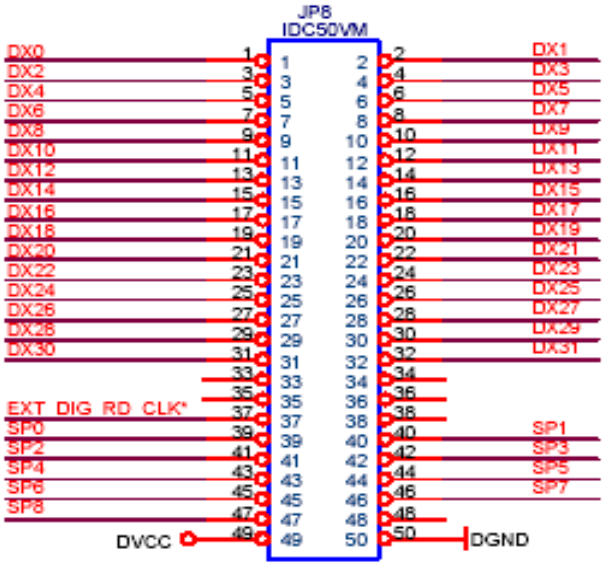
The following table gives the pin numbers and functions for the JP5 (OMNIBUS slot 0) and JP9 (OMNIBUS slot 1) connectors.

Pin Number	JP5 Function	JP9 Function	Direction (from M6713)
1, 3-6	Address bus 9-13		O
2, 19, 20, 49, 50	Digital ground		O, power
7, 9, 11-13, 15, 17	Reserved	Reserved	NA
8, 10	Digital +3.3V		Power
14	Module trigger 0		O
16	Module trigger 1		O
18	Module trigger 2		O
21	Processor timer channel 1		O
22	Module trigger in	External trigger 1	O
23,25	Analog +15V (OMNIBUS +12V)		O, power
24	CLKS0	CLKS1	I
26	CLKR0	CLKR1	I/O
27	FSR0	FSR1	I/O
28	CLKX0	CLKX1	I/O
29	External interrupt 1	External interrupt 3	I
30	DR0	DR1	I
31	FSX0	FSX1	I/O
32	DX0	DX1	O
33-48	Data bus 16-31		I/O

TABLE 39. I/O Module Bus Connectors

JP8 – Digital I/O Connector

Connector type:	0.1” double row shrouded header, center bump polarized, Thomas and Betts 609-5027
Number of pins:	50
Mating connector:	AMP 1-746285-0



The following table gives the pin numbers and functions for the digital IO connector.

Pin Number	JP8 Function	Direction (from M6713)
1-32	Digital I/O bit 0..31	I/O
33-36	Not used	-
37	External Digital Readback Clock(latch falling edge)	I
38	Not used	-
39-47	Spare pins from PCI FPGA	-
49	DVCC (digital +5 V)	Power
50	DGND (digital ground)	Power

TABLE 40. Digital I/O Connector

JH1 – FPDP Transmit Port Connector

Connector type: P50E-080P1-S1-TG

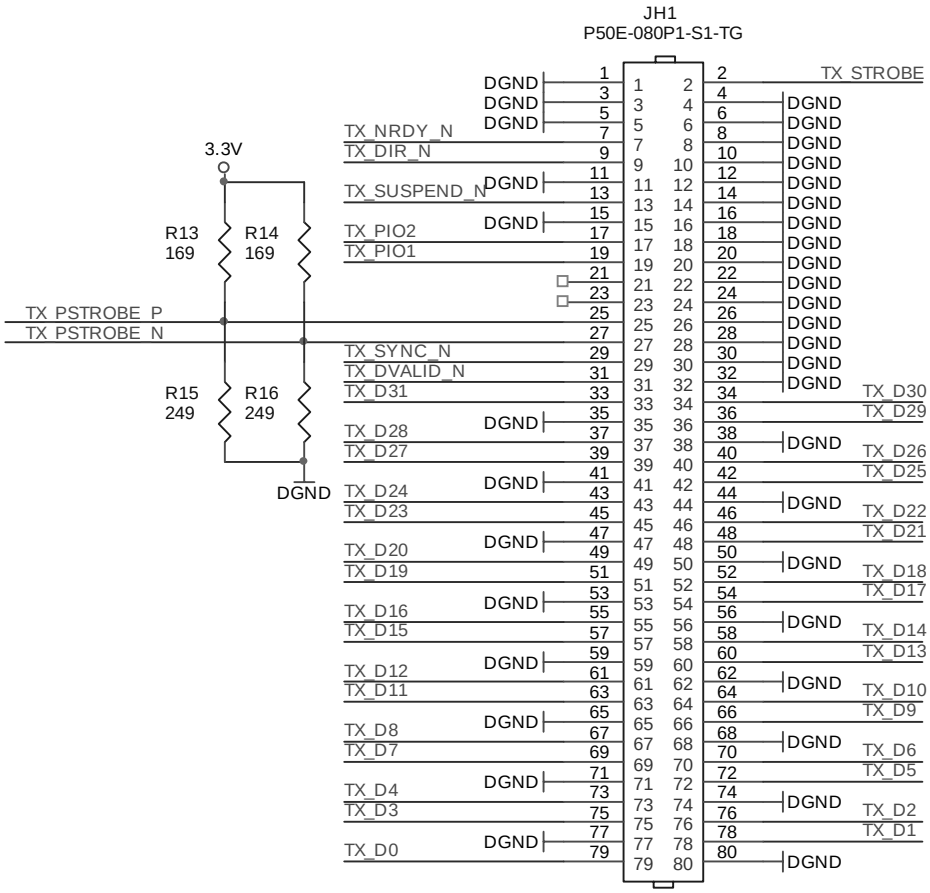
Number of pins: 80

Mating connector: P25E-080S-TGF (Manuf.3M)

The following table gives the pin numbers and functions for the JH1 connector.

Pin Number	JH1 Function	Direction (from M6713)
1,3,4,5,6,8,10, 12,14,16,18,20 ,22,24,26,28, 30,32,35,38,41 ,44,47,50,53, 56,59,62,65,68 ,71,74,77,80	Digital Ground	Power
2	Tx Strobe	O
7	Tx NRDY N	O
9	Tx DIR N	O
13	Tx Suspend N	O
17	Tx PIO2	O
19	Tx PIO1	O
25	Tx PStrobe P	O
27	Tx PStrobe N	O
29	Tx Sync N	O
31	Tx DValid N	O
33, 34, 36, 37	TxD31, TxD30, TxD29, TxD28	O
39, 40, 42, 43	TxD27, TxD26, TxD25, TxD24	O
45, 46, 48, 49	TxD23, TxD22, TxD21, TxD20	O
51, 52, 54, 55	TxD19, TxD18, TxD17, TxD16	O
57, 58, 60, 61	TxD15, TxD14, TxD13, TxD12	O
63, 64, 66, 67	TxD11, TxD10 , TxD9 , TxD8	O
69, 70, 72, 73	TxD7 , TxD6 , TxD5 , TxD4	O
75, 76, 78, 79	TxD3 , Tx2D , TxD1,	O

TABLE 41. FPDP JH1 Tx Port Connector



JH2 – FPDP Receive Port Connector

Connector type: P50E-080P1-S1-TG

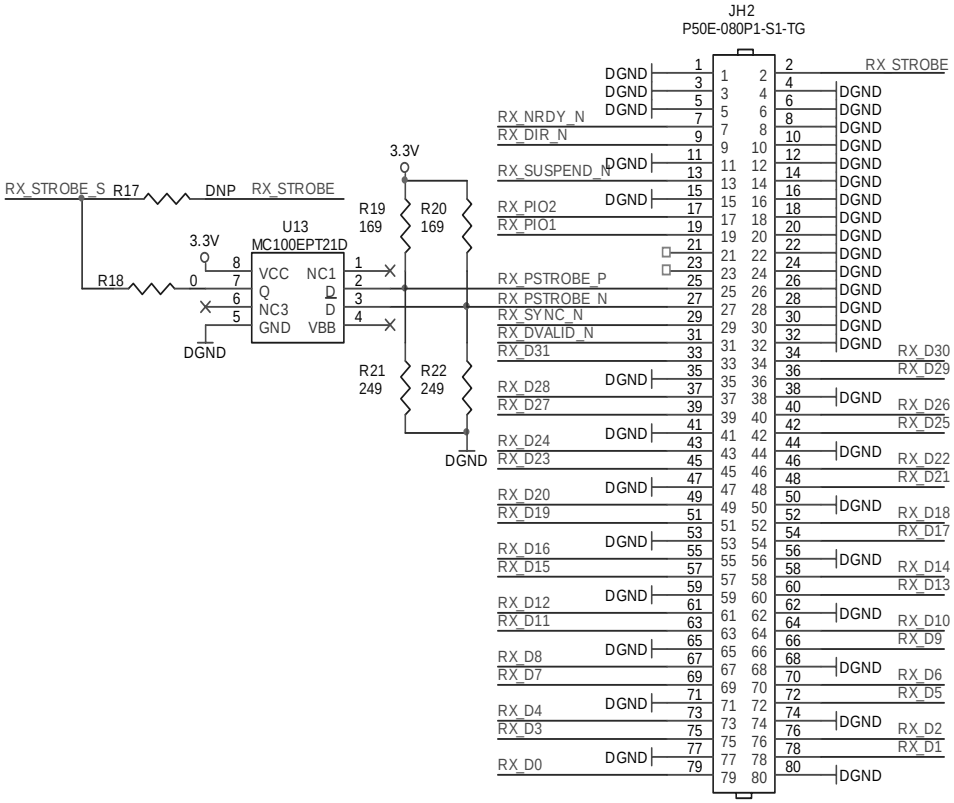
Number of pins: 80

Mating connector: P25E-080S-TGF (Manuf.3M)

The following table gives the pin numbers and functions for the JH2 connector.

Pin Number	JH2 Function	Direction (from M6713)
1,3,4,5,6,8,10, 12,14,16,18,20 ,22,24,26,28, 30,32,35,38,41 ,44,47,50,53, 56,59,62,65,68 ,71,74,77,80	Digital Ground	Power
2	Rx Strobe	I
7	Rx NRDY N	I
9	Rx DIR N	I
13	Rx Suspend N	I
17	Rx PIO2	I
19	Rx PIO1	I
25	Rx PStrobe P	I
27	Rx PStrobe N	I
29	Rx Sync N	I
31	Rx DValid N	I
33, 34, 36, 37	RxD31, RxD30, RxD29, RxD28	I
39, 40, 42, 43	RxD27, RxD26, RxD25, RxD24	I
45, 46, 48, 49	RxD23, RxD22, RxD21, RxD20	I
51, 52, 54, 55	RxD19, RxD18, RxD17, RxD16	I
57, 58, 60, 61	RxD15, RxD14, TxD13, RxD12	I
63, 64, 66, 67	RxD11, RxD10, RxD9 , RxD8	I
69, 70, 72, 73	RxD7 , RxD6 , RxD5 , RxD4	I
75, 76, 78, 79	RxD3 , Rx2D , RxD1,	I

TABLE 42. FPDP JH2 Rx Port Connector



JP12 - SyncLink/ClkLink

The SyncLink connector allows M6713 to synchronize to external hardware or other Innovative cards.

Connector type: 0.1” double-row shrouded header

Number of pins: 14

Mating connector: AMP

The following table gives the pin numbers and functions for the SyncLink/ClockLink connector.

Pin Number	JP12 Function	Direction (from M6713)
1	Clocklink Out +	O
2	Clocklink Out -	O
3	Clocklink In +	I
4	Clocklink In -	I
5	Synclink bus pin 2	I/O
6	Digital ground	Power
7	Synclink bus pin 1	I/O
8	Digital ground	Power
9	Synclink bus pin 0	I/O
10	Digital ground	Power
11	Synclink bus pin 3	
12	Synclink bus pin 5	
13	Synclink bus pin 4	
14	NC	-

TABLE 43. SyncLink Connector

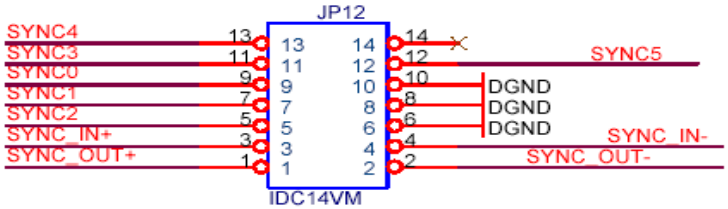


FIGURE 17. JP12 SyncLink Connector Pin Orientation

JP14, JP15 – Processor Serial Port Connectors

Connector type:	2 mm double row header
Number of pins:	10
Mating connector:	Samtec SQT style (for board-board applications)

The following table gives the pin numbers and functions for the JP14 (McBSP 0) and JP15 (McBSP 1) connectors. Pin functions of JP14 are identical to those of JP15 except where noted.

Pin Number	JP14 Function	JP15 Function	Direction (from M6713)
1	CLKS0	CLKS1	I
2	FSR0	FSR1	I/O
3	CLKR0	CLKR1	I/O
4	FSX0	FSX1	I/O
5	CLKX0	CLKX1	I/O
6	Digital 3.3V	Digital 3.3V	Power
7	DR0	DR1	I
8	Digital 5V	Digital 5V	Power
9	DX0	DX1	O
10	Digital Ground	Digital Ground	Power

TABLE 44. DSP Serial Port Connector

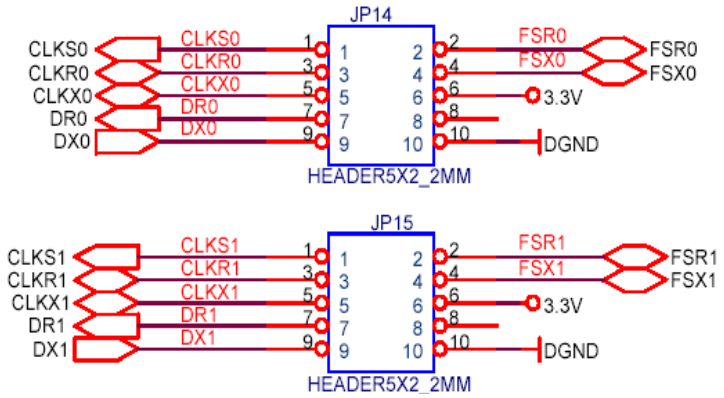


FIGURE 18. JP14, JP15 DSP Serial Port Connector

JP16 – JTAG Debugger Connector

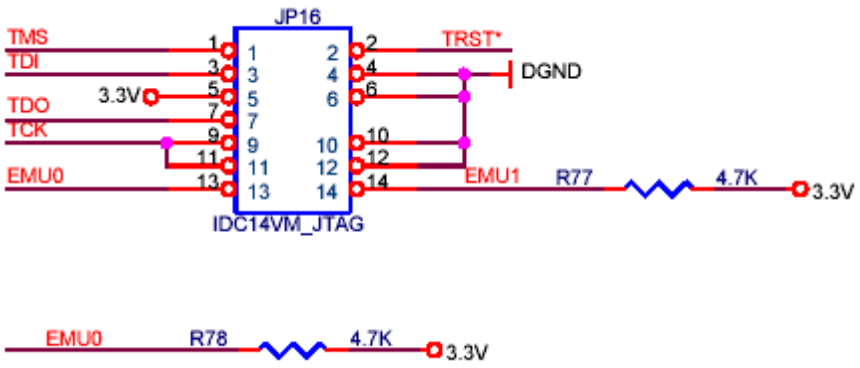
Connector type:	Shrouded header, pin 6 removed for key
Number of pins:	14
Mating connector:	AMP 746285-2

The following table gives the pin numbers and functions for the DSP JTAG connector.

Pin Number	JP16 Function	Direction (from M6713)
1	TMS	I
2	TRST*	I
3	TDI	I
5	Digital +3V	Power
7	TDO	O
9,11	TCK	I
13	EMU0	I/O
14	EMU1	I/O
4, 6, 8, 10, 12	Digital ground	Power

TABLE 45. DSP JTAG Debugger Connector

FIGURE 19. JP16 DSP JTAG Debugger Connector



JP18 – Power Test Connector

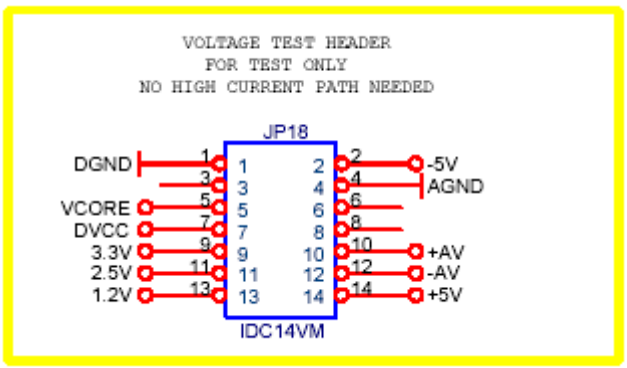
Connector type:	Shrouded header
Number of pins:	14
Mating connector:	AMP 746285-2

This connector is only for debug. Normally used for production testing.

Pin Number	JP18 Function
1	DGND
2	-5V
3	-
4	AGND
5	DSP Core Voltage (1.4V)
6	-
7	5V
8	-
9	3.3V
10	+AV (+15)
11	2.5V
12	-AV (-15V)
13	1.2V (Spartan3 FPGA Core)
14	+5V

TABLE 46. Power Test Connector

FIGURE 20. JP18 Power Test Connector



JP17 – Interface Logic (Spartan3) JTAG Connector

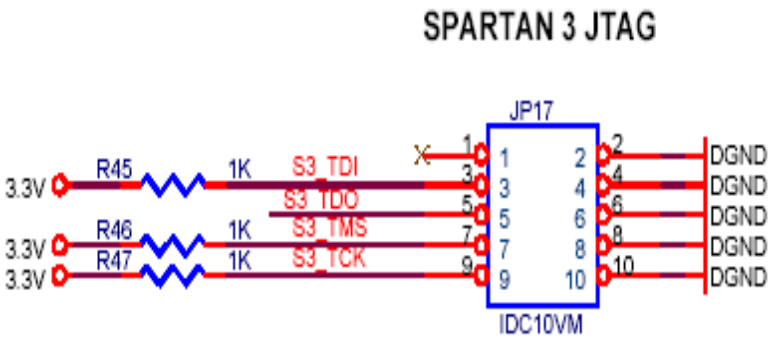
Connector type:	2MM unshrouded header
Number of pins:	10
Mating connector:	Samtec SQT style (for board-board applications)

This connector is for programming and developing logic for the Spartan3 Interface logic FPGA.

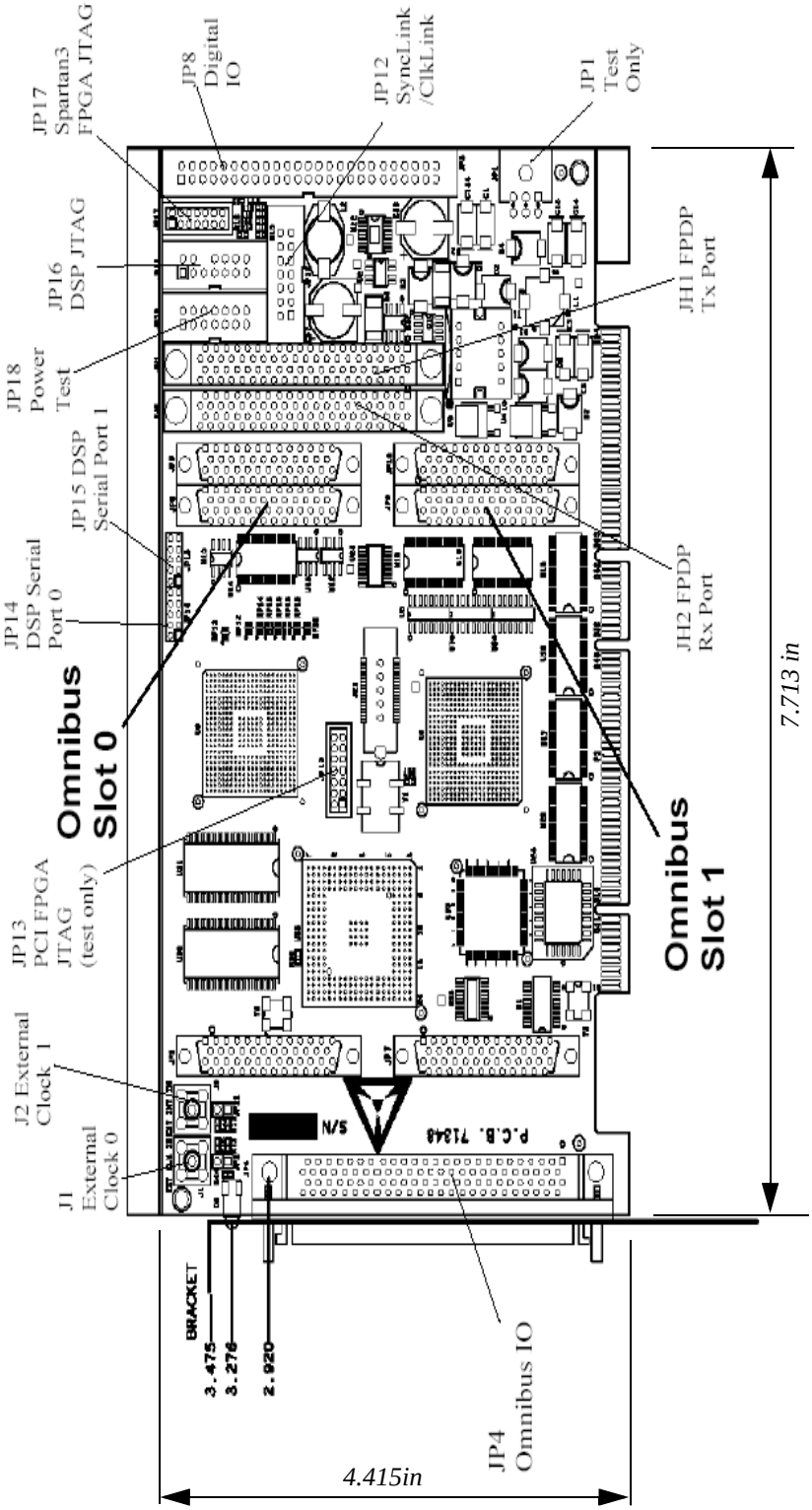
Pin Number	JP17 Function
1	-
2	DGND
3	TDI
4	DGND
5	TDO
6	DGND
7	TMS
8	DGND
9	TCK
10	DGND

TABLE 47. JP17 Interface Logic (Spartan3) JTAG Connector

FIGURE 21. JP17 Interface Logic (Spartan3) JTAG Connector



Board Layout Drawing (Rev B)



Symbols
.c file 58

A
Applets
  BinView 92

B
BinView 92
Borland C/C++ 58

C
C++ Builder 11

E
edit-compile-test cycle 58
E-Mail 13
errors 59

I
Innovative Integration
  E-Mail 13
  Technical Support 12
  Web Site 12, 13

L
Library Code 61

M
Matador 11

T
target applications 58
Technical Support 12
Troubleshooting 133